Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2011

Linda Maria Velte

Repositório de registos electrónicos de saúde baseado em OpenEHR

Electronic Health Record Repository based on the OpenEHR standard

Linda Maria Velte

Repositório de registos electrónicos de saúde baseado em OpenEHR

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor José Luís Guimarães Oliveira e do Doutor Carlos Manuel Azevedo Costa, Professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president	**Doutor Joaquim Arnaldo Carvalho Martins** Prof. Catedrático da Universidade de Aveiro
vogais / examiners committee	**Doutor Rui Pedro Sanches de Castro Lopes** Prof. Coordenador do Dep. Informática e Comunicações da ESTG do Instituto Politécnicno de Bragança
	Doutor José Luis Guimarães Oliveira Professor Associado da Universidade de Aveiro (orientador)
	Doutor Carlos Manuel Azevedo Costa Professor Auxiliar da Universidade de Aveiro (co-orientador)

Acknowledgements

I would like to thank my parents for their love, for providing for my education and for always encouraging me.

I also thank my friends and my two sisters for their support and care and for being there at all moments.

Finally I thank all my professors at the University of Aveiro who significantly contributed for my personal and academic education.

Resumo

Um registo electrónico de saúde agrega toda a informação médica relevante de um paciente, permitindo uma filosofia de armazenamento orientada ao mesmo. Desta forma todo o historial médico do paciente encontra-se armazenado num único registo, permitindo a optimização de custos e tempo gasto nas diferentes tarefas, através de partilha de informação entre diferentes instituições médicas. Para possibilitar esta partilha é necessário definir um formato comum em que a informação é armazenada. Para tal foram definidas diversas normas que ditam as regras de armazenamento, troca e recuperação de informação médica. Uma destas normas é o Open Electronic Health Record (OpenEHR).

O objectivo desta dissertação é criar um repositório que permite o armazenamento de registos médicos que sigam a norma OpenEHR. A implementação dá origem a três componentes de software, sendo eles uma base de dados Extensible Markup Language (XML) para armazenamento de registos médicos, um conjunto de serviços para gestão e pesquisa da informação armazenada e uma interface web para demonstração das funcionalidades implementadas.

Abstract

An Electronic Health Record (EHR) aggregates all relevant medical information regarding a single patient, allowing a patient centric storage approach. This way the complete medical history of a patient is stored together in one record, making it possible to save time and work by allowing the sharing of information between health care institutions. To make this sharing possible there has to be agreed on the format in which the information is saved. There are many standards to define the way health information is stored, exchanged and retrieved. One of this standards is the Open Electronic Health Record (OpenEHR).

The goal of this thesis is to create a repository which allows to store and manage patient records which follow the OpenEHR standard. The result of the implementation consists in three software parts, being them a Extensible Markup Language (XML) repository to store health information, a set of services allowing to manage and query the information stored and a web interface to demonstrate the implemented functionalities.

Contents

List of Figures

List of Tables

Acronyms

ACR American College of Radiology. 17

ADL Archetype Definition Language. 9, 10, 35

AM Archetype Model. 9, 10

AOM Archetype Object Model. 10, 35

API Application Programming Interface. 10, 21, 26–28, 35, 39, 40, 42, 64

AQL Archetype Query Language. 9

CDA Clinical Document Architecture. 7, 11, 12, 14, 18

CEN European Committee for Standardization. 16

CLOB Character Large Object. 25

CSS Cascading Style Sheets. 45, 46

DICOM Digital Imaging and Communications in Medicine. 7, 13–19

DOM Document Object Model. 28, 44, 46

ebXML Electronic Business using Extensible Markup Language. 15

EHR Electronic Health Record. 1, 5–7, 10, 12–19, 24, 27, 32, 35, 37, 42–44, 48, 50, 55, 60, 61, 64

GIF Graphical Interchange Format. 19

GPL General Public License. 34

GWT Google Web Toolkit. 44–46

HIMMS Healthcare Information and Management System Society. 1

HL7 Health Level Seven. 7, 10–12, 14, 15, 17–19, 64

HTML Hypertext Markup Language. 13, 24, 25, 44–49

HTTP Hypertext Transfer Protocol. 13, 14, 24, 27, 39, 40, 42

IHE Integrating the Healthcare Enterprise. 14, 15, 19

IM Information Model. 9

ISO International Organization for Standardization. 13, 44

ISORM International Standards Organization Reference Model for Network Communications. 17

JPEG Joint Photographic Experts Group. 13, 14, 19

JSNI JavaScript Native Interface. 45

JSON JavaScript Object Notation. 45, 48

LGPL Lesser General Public License. 34, 45

MIM Message Information Model. 11

MML Medical Markup Language. 7, 12, 18

MPL Mozilla Public License. 34

MVC Model View Controller. 45

NEMA National Electrical Manufacturers Association. 17

ODMG Object Data Management Group. 25

OpenEHR Open Electronic Health Record. 1–3, 7–10, 16, 18, 21, 22, 24, 27, 30, 32–35, 37, 49, 50, 55, 63, 64

OQL Object Query Language. 25

REST Representational State Transfer. 22, 24, 27, 39, 40, 64

RID Retrieve Information for Display. 7, 12, 14, 19

RIM Reference Information Model. 11, 35

RM Reference Model. 9

RPC Remote Procedure Call. 24, 45

SM Service Model. 10

SOA Service Oriented Architecture. 22, 64

SOAP Simple Object Access Protocol. 22, 24, 41

SQL Structured Query Language. 25, 26

SR Structured Report. 7, 16–18

UID Unique Identifier. 14

URL Uniform Resource Loader. 24, 39, 40, 46, 51

W3C Worl Wide Web Consortium. 27, 42

WADO Web Access to DICOM Persistent Objects. 7, 12, 13, 18, 19

WSDL Web Service Definition Language. 14, 24, 41

XDS Cross-Enterprise Document Sharing. 7, 12, 14, 15, 19

XHTML Extensible Hypertext Markup Language. 14, 46

XML Extensible Markup Language. 1, 9, 11–13, 24–30, 32, 35, 37, 39–44, 46, 48–51, 54, 60, 62–64

XSD XML Schema Definition. 26, 35, 50

XSL Extensible Stylesheet Language. 26

XSLT Extensible Stylesheet Language Transformations. 26, 39

XUL XML User Interface Language. 45

ZUML ZK User Interface Markup Language. 45–48

Chapter 1

Introduction

1.1 Motivation

According to the Healthcare Information and Management System Society (HIMMS) an Electronic Health Record (EHR) is a longitudinal electronic record of patient health information generated by one or more encounters in any care delivery system [1]. Included in this information are patient demographics, progress notes, problems, medications, vital signs, past medical history, immunizations, laboratory data and radiology reports.

Before EHRs the patients information was dispersed over different applications. Each area (such as Radiology or Lab, etc) had their own specific application to store relevant information. A clinical user had to open all the applications, log in and search for the patients record within each of them to get the whole record. An EHR allows an integrated access to the patients information as it gives the possibility to aggregate all medical data of the patient, and consequently, to have a patient centric storage approach. This way, the complete medical history of a patient is stored together in one record, making it possible to save time and work. For instance, a patient who had medical exams in one hospital does not have to remake them if he needs the results in another hospital, in another city. The information is associated to the patient and not to the institution where it was originated.

To make this sharing of information possible there has to be agreed on the format in which the information is saved. There are many standards to define the way health information is stored, exchanged and retrieved. One of this standards is the Open Electronic Health Record (OpenEHR) [2].

OpenEHR follows a two-level modelling approach, dividing information from knowledge. This fact has revolutionized the way health information systems are developed. The medical area is always changing, so it is very difficult to define one representation for information that is valid for every single data that might be stored. The proposed separation by the two-level modelling concept makes it possible to have a division between the information that is stored and the rules that constrain it. This constraint is made using archetypes, which are the set of rules that dictate which information is valid. Domain experts define all record types accepted (making it possible to create new ones every time it is necessary), while the system users are just concerned about creating instances that represent the data as described by the archetypes. The OpenEHR project is not a ready to use information system for health record management but about creating specifications, open software and tools that allow to develop such a system.

1.2 Goals

The main goal of this thesis is to create a repository which allows to store and manage patient records which follow the OpenEHR standard (OpenEHR reference model instances). This means it should be possible to insert, visualize and query health information, following the format dictated by the standard.

The result of the implementation should consist in three software parts: a repository to store the health information, a set of services allowing to manage and query the information stored and a web interface to demonstrate the implemented functionalities. Before any implementation can be done, the first objective must be a full comprehension of the OpenEHR standard, including its architecture, the format of its records and the nature of the information that will be stored. This knowledge will allow the design of a solution that fits the needs of health information repository, respecting the OpenEHR standard in a way that any other system can make used of it, knowing only the standards published rules.

After knowing the OpenEHR standard and having considered the technologies available, it will be designed a repository capable of handling this kind of records. The goal is to have a repository efficient and flexible enough to be integrated in a health information system. This means to study the requisites of such a system and evaluate the technologies in terms of if they can fulfil them. The implementation of the repository will result in a database capable of storing electronic health records according to the OpenEHR standard.

The second part of the implementation focuses on the communication between the repository and the applications in which it will be integrated. The goal is to create a business layer which allows the applications to access the repository. To achieve that, there will be created a set of services that cover the functionalities required by a health information system. The services must be accessible in a normalized way to allow interoperability with all kinds of applications, independently of the platform. This layer, together with the repository itself forms the main part of the final implementation. The intention is to create a software component that is flexible and ready to use by many different kinds of applications for different purposes.

For demonstrative reasons a third part will be created: a web interface. This interface has the goal to make the functionalities implemented visible, being more a demo of the possible usage of the repository than an important part of the implementation. The interface will show the results obtained by using the offered services in a graphic way, trying to represent the normal usage scenarios of this kind of repositories.

Another important part of this thesis is the academic aspect of the work. Many technologies will be studied and compared, from electronic health record standards to storage technologies and web development frameworks. The final implementation will also be evaluated in terms of performance and possible usages, making it interesting for everyone who needs to choose a software component to perform similar operations. This evaluation will refer to the advantages and disadvantages of the created application, considering different usage scenarios and testing the performance with sets of test data, created for this purpose.

1.3 Outline

This document is divided into three main chapters: Electronic Health Records, OpenEHR Repository and Evaluation. Each of this chapters is divided into several sections which are listed below.

- **Chapter 2: Electronic Health Records** This Chapter discusses the main Electronic Health Record standards, dividing them into content format and message standards. It ends with a comparative analysis of the standards explained, still respecting the proposed division.
- **Chapter 3: Planning and Development of an OpenEHR Repository** The implemented repository is described in this chapter. It starts with an overview on how the repository fits within the OpenEHR architecture. Next is described the adopted architecture including the format of the records stored in the database and the logical layers in which the architecture can be divided: core, service and application layer. Each of this layers is explained separately in its own section. The core layer (Storage of OpenEHR Records section) focuses on the database technologies studied, the database software used and how the database was implemented. The service layer section explains which services were implemented and what functionalities they support, including as well a description of the API used to connect to the database. The application layer section is about the implemented web interface, explaining the considered technologies and some aspects of the implementation. The chapter ends with a section containing the results of the implementation.
- **Chapter 4: Evaluation** This chapter explains which tests were made to the final implementation result, what was done to improve performance and which are the ideal utilization scenarios of this repository.

Chapter 2

Electronic Health Records

Electronic Health Record (EHR) was defined in 1998 by Iakovidis as "digitally stored health care information about an individual's lifetime with the purpose of supporting continuity of care, education and research, and ensuring confidentiality at all times" [3]. The EHR includes information such as patient demographic, progress notes, problems, medications, vital signs, post medical history, radiology, reports. This information can be generated by one or more encounters in any care delivery system.

In non EHR systems, as can be visualized in figure 2.1, each organization has a system to capture patient data for their specific area. The provider must open each application to view the specific data. This data may or may not be in conformance with a standard. This turns the consult of a patients medical history into a difficult process. For instance to view a lab result it is necessary to open the lab application, to consult a radiology exam it is necessary to access the radiology application, and the same for other scenario cases. This it not only a time consuming process for the medical operator but also results in a dispersion of the medical history of the patient.

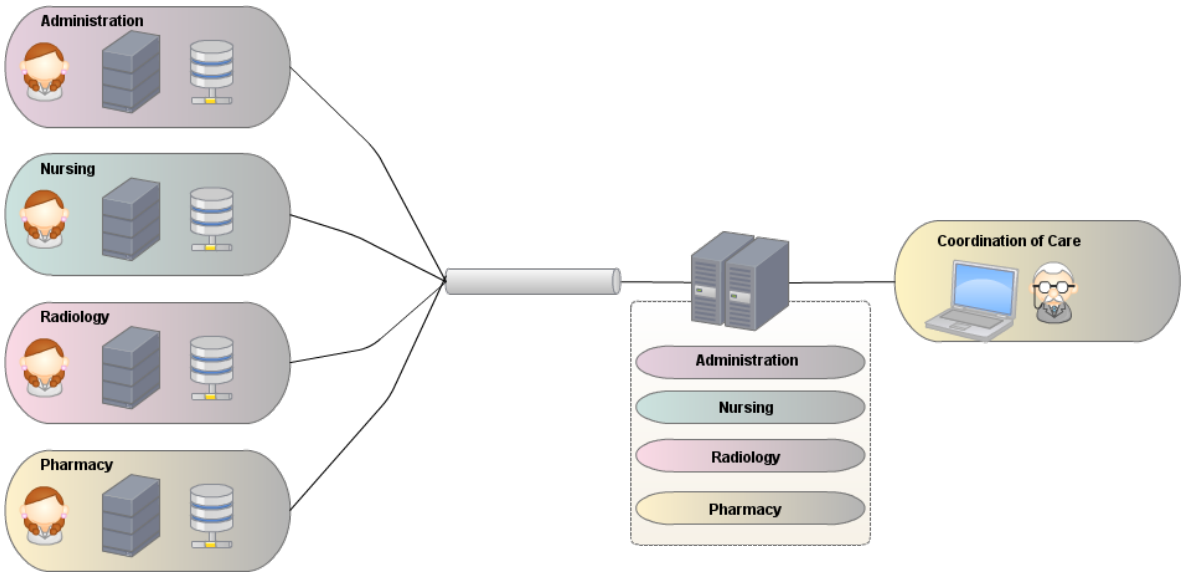

Figure 2.1: EHR Definition - Pre EHR Healthcare System adapted from [4]

EHR represents the integration of health care data from a participating collection of systems for a single patient (figure 2.2). The great advantage of this kind of system is its patient centric nature. When a patient goes to see a doctor there will be often the need to consult his medical history. Having all information gathered in one record facilitates the process and can also save lots of money for the hospital, as for instance there is no need to repeat a medical exam if it was made previously in a different medical institution.

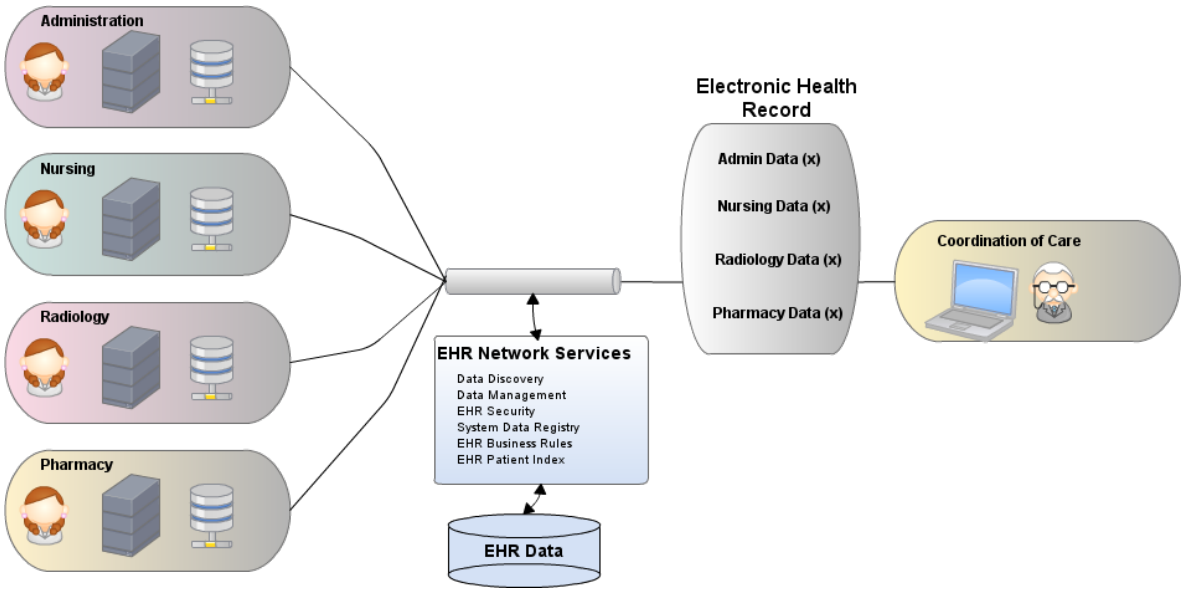

Figure 2.2: EHR Definition - EHR Healthcare System adapted from [4]

As an effort to solve the interoperability issues there are several standards under development. A standard is very important to assure that it is possible to share medical information between different institutions. A patient's EHR must be accessible and the information understandable, no mater what hospital/medical institution he visits. To accomplish that there must an agreement on the language that is "spoken". This is done via standards that define not only how the information is structured and represented but also how it can be retrieved and shared between systems. To allow a better understanding of the aspects to have in mind when implementing a health information system this chapter discusses a few standards, being only presented the most popular ones.

Before looking at existing standards it is necessary to understand some definitions. Because the health area is always evolving there are often new kinds of information that need to be stored, which means it is necessary to have a very flexible data model that can easily be changed and/or extended. What is needed is an abstraction of the database structure, which means that the goal is to have a storage model that is capable of accepting data instances no matter what format they have. With other words, a health information system needs a dynamic database model which adapts to the information and not the other way around. To accomplish that most standards use a two level modelling approach, which means a separation of information from knowledge. This methodology results in a reference model which specifies how the information is structured and in a set of constrain rules which dictate what data types, structures and values are valid. This rules can be archetypes or templates. Archetypes are prototypes which define which format the information can follow, having sig-

nificantly more expression power than fixed templates [5]. An archetype can be defined as a set of guidelines which, when followed, lead to valid information instances. This way a record has always the same meaning, no matter where it is used, allowing for great interoperability between healthcare systems.

The standards that will be discussed can be divided into content format standards (OpenEHR, Health Level Seven (HL7) Clinical Document Architecture (CDA) and Medical Markup Language (MML)) and communication standards (Web Access to DICOM Persistent Objects (WADO), Retrieve Information for Display (RID) and Cross-Enterprise Document Sharing (XDS)). Additionally, there will be described two standards that fulfil the characteristics of both of these groups (EHRcom and Digital Imaging and Communications in Medicine (DICOM) Structured Report (SR)) therefore they will be described in a separate section called global standards.

2.1 Content Format Standards

Content Format Standards define how information should be stored, defining the format of the content contained by the EHR. In this group are included OpenEHR, HL7 CDA and MML.

2.1.1 OpenEHR

The OpenEHR foundation [1] is an international foundation with the goal to improve interoperability and electronic health record management. It supports the open research, development and implementation of OpenEHR electronic health records [6].

OpenEHR is an open standard specification in health informatics that describes the management, storage, retrieval and exchange of data in EHR's. In OpenEHR all data for a person is stored in a "one lifetime", vendor independent, person centred EHR. The OpenEHR specification is not concerned with the exchange of data between EHR-systems as this is the primary focus of message standards such as ISO 13606 and HL7. This standard has the goal to create high-quality, re-usable clinical models of content and process (known as archetypes) and to enable semantic interoperability between and within EHR systems. As long as other standards focus on one specific area, such as DICOM on digital imaging or HL7 on patient management, the OpenEHR standard can be used to describe any kind of data due to the two level modelling approach adopted, since any information structure can be defined with help of an archetype.

Two Level Modelling

The goal of the OpenEHR standard is to achieve semantic interoperability regarding the whole health area. For this to be possible, a generic, dynamic health information model is needed, capable of storing new kinds of data without having to change the data model. The drawback of this kind of model is that there is no control about what information is stored, leading to low data quality and being not very different from a unstructured model. This issue is solved by the two level modelling approach by creating a second level used to constrain the information [7]. The two level modelling methodology consists basically in the

[1] http://www.openEHR.org/home.html

separation of information and knowledge. The result is a generic health record model that enables the storage of a wide variety of health information. The structure of the information is constrained by the the archetype model. This way domain specific knowledge is independent of the implementation, which allows to have different views of the same knowledge. This methodology leads to a generic model capable of storing any kind of health records which means a great interoperability improvement, but at the same time makes it possible to control the structure of the information using archetypes.

Archetypes

An archetype represents a clinical concept. It is used to constrain instances of the OpenEHR information model by defining a valid structure, data types and values. Further than that an archetype is a clinically meaningful entity. An electronic health record which has been archetyped will have the same meaning no mater where it appears, so it can be shared by multiple health systems [8] . As shown in figure 2.3, archetypes are defined by domain experts. This way the user who creates and inserts the information has not to know how the information must be structured, as it will be validated against the rules defined by the domain experts (in this case medical professionals).

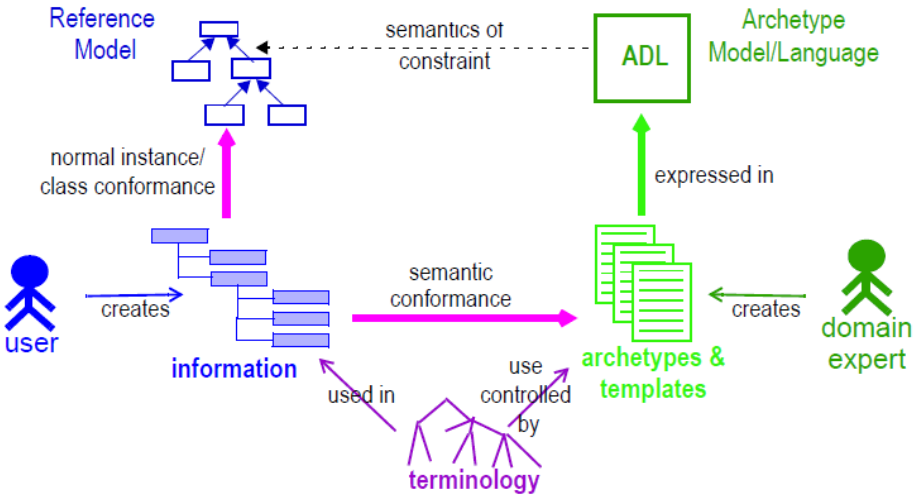

Figure 2.3: EHR Standards - OpenEHR Two Level Modelling Approach from [9]

An archetype description consists basically of three parts: descriptive data, constraint

rules and ontological definitions. The descriptive data consists in an identification of the archetype (ID), a description of what clinical concept is modelled by the archetype and metadata such as author, version and purpose of the archetype. Figure 2.4 shows the header information of a transfusion entry.

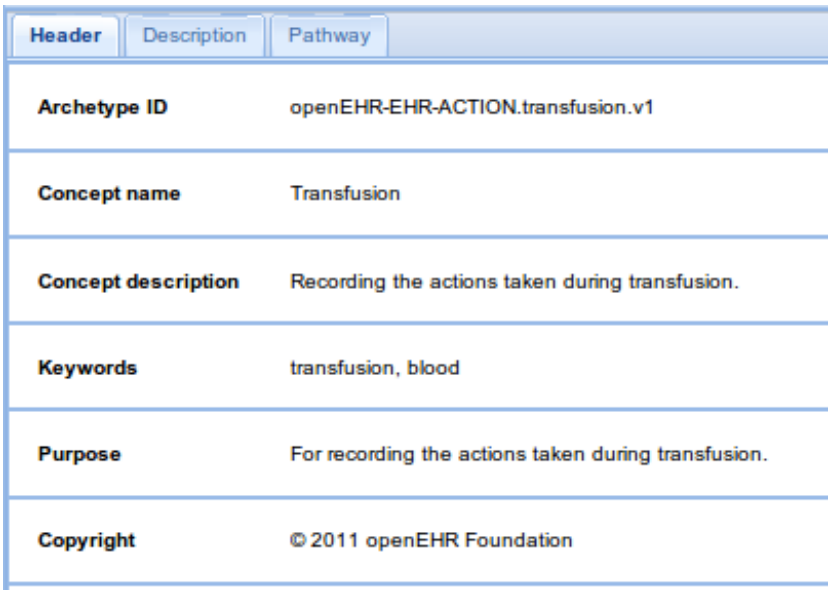

Figure 2.4: EHR Standards - OpenEHR Transfusion Entry Header

The constraint rules form the main part of the archetype, since they specify which structure, cardinality and content is valid in an OpenEHR Reference Model instance describing the clinical concept related to the archetype. The ontology definitions define control vocabulary, in machine readable codes, which can be used in specific parts of an archetype instance [2]. Archetypes are defined using the Archetype Definition Language (ADL) [10], which is a formal language related to the reference model, since it allows to define archetypes which constrain the reference model. To query OpenEHR information defined by archetypes it was defined the Archetype Query Language (AQL) which allows to access the nodes similar to a XPath query on an Extensible Markup Language (XML) document [11]. OpenEHR maintains an online repository of defined archetype and templates, with the possibility to export them in ADL or XML format, called the OpenEHR knowledge base [2].

OpenEHR Architecture

The OpenEHR architecture follows the package structure depicted in figure 2.5. The reference model allows data interoperability, because data is exchanged between systems only in terms of standard open reference model instances. Semantic interoperability is achieved by sharing archetypes. Archetypes are used at both ends: to validate and to query the data. The OpenEHR Reference Model (RM) defines the structure and semantics of information. The RM corresponds to the information viewpoint and defines the data of OpenEHR Electronic Health Record systems [12]. The Information Model (IM) is designed to be constant in a long term, to minimize the need for software and schema updates. The Archetype Model (AM)

[2]http://www.openEHR.org/knowledge/

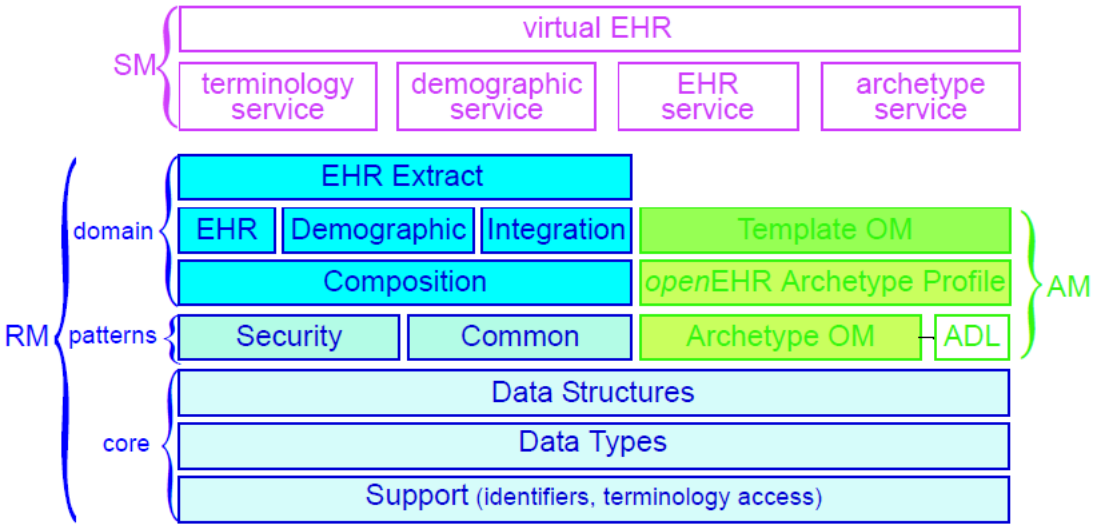

Figure 2.5: EHR Standards - OpenEHR Package Structure from [9]

defines the structure and semantics of archetypes and templates. The AM consists of the ADL, the Archetype Object Model (AOM) and the OpenEHR Archetype Profile [5]. Service Model (SM) includes definitions of basic services in the health information environment, centred around the EHR.

Projects using OpenEHR

The OpenEHR specification was implemented by a team from Sweden using Java programming language. This implementation was donated to the OpenEHR foundation and adopted as the OpenEHR Java Reference Implementation in 2005 [12]. There is a fully implemented version of the reference and archetype models. Many organisations are using/contributing to the OpenEHR standard. Two important references are Opereffa [3] and Kanolab [4]. Opereffa stands for OpenEHR Reference Framework and Application and it is a project for creating an open source clinical application which will be driven by the Clinical Review Board of OpenEHR. Kanolab, from Wased University, has been developing final products and tools, Application Programming Interface (API)'s and techniques using the OpenEHR standard. An example is the generic EHR processing platform openEHRApp [5].

2.1.2 HL7

HL7 organisation was formed in the United States in 1987. HL7 is a not-for-profit volunteer organization that creates standards for the exchange, management, and integration of electronic healthcare information and the management, delivery, and evaluation of healthcare services [13]. HL7 is based on the idea that an event causes the exchange of messages between a pair of applications (trigger event). For the various trigger events are defined messages, being a message send to the receiving unit when the specific event associated to the message

[3] http://opereffa.chime.ucl.ac.uk
[4] http://www.kanolab.info
[5] http://kenai.com/projects/openEHR-app

occurs [14]. There are defined several types of trigger evens, such as admission of a new patient, entering of an order or to inform that test results are ready. For example when a patient is admitted the application managing patients data informs the other applications of the new demographic data, sending an admission message. These messages are unsolicited updates and need to be confirmed with an acknowledge message from the receiving applications [15]. Also supported are queries, which can be send by an application needing some data to continue processing.

The HL7 message protocol is the most implemented in the healthcare area, offering a standard for message exchange between applications sharing medical information. However it is no guarantee for interoperability because it does not describe how the underlying information model should be defined. On the other hand this fact allows more flexibility. To solve the interoperability issues verified in earlier versions, version 3 started to be developed in 1994 and defines 4 models: use case model, information model, interaction model and message design model. The use case model defines the possible scenarios, specifying how the information must flow between applications and how this information is processed by them. The information model defines how information should be structured, defining a shared model which every HL7 message has to respect: the object-oriented Reference Information Model (RIM). The RIM is the reference for class and attribute definitions and represents the clinical data and the connections that exist between the information carried in the fields of HL7 messages. To make use of this new model, version 3 introduces the CDA). The interaction model defines how systems should communicate with each other using HL7 messages. It is in this model that the trigger events and associated messages are defined, being an interaction not more than a association between a trigger event, a message and two applications. Finally, the message design model defines the message format, having in sight the classes defined by the RIM and taking in consideration the attributes required for a specific interaction. The Message Information Model (MIM) is a subset of the RIM and provides the basis for constructing a message, containing the classes, attributes and connections needed for a particular set of messages [16].

Clinical Document Architecture

The HL7 CDA is a document mark-up standard that specifies the structure and semantics of clinical documents for the purpose of exchange [17]. The CDA consists of 3 levels and each of them takes the mark up of the previous level and adds more mark up to compose a clinical document. However, this does not change the clinical content of the document [18]. CDA documents are encoded in XML with a semantic structure and derive their meaning from the HL7 RIM. HL7 brings an improvement to interoperability by being not only human readable and independent from the database implementation, but also by allowing both structured and non structured information in the same document. CDA provides an exchange model for clinical documents, offering a great freedom in structuring a document. The great advantage of CDA is being highly structured and modular at the same time. The structure defined by CDA consists in a division of the document in two parts: the header and the body. The header contains administrative information such as the destination of the document, version control fields, service information and information to identify the document, allowing the clinical document exchange and management. It contains also information about the author of the document, which is important for legal reasons. The body of the document can be structured (for a specific and complete exchange of clinical data structure) or unstructured (can be used

in a huge number of different circumstances) and contains the clinical information that needs to be exchanged [19].

2.1.3 Medical Markup Language

MML was developed to enable the exchange of documents between different medical information providers. This standard has been evolving since its introduction in 1999 (this version used XML as a metalanguage). The latest version of MML (version 3.0) is based on the HL7 CDA with extended functions. Version 3.0 consists in 14 modules that are classified into three categories: MML header module, MML document information module and MML content modules. MML header includes the information for data transmission, while MML body includes several module items. One module item contains only one module and one document information. The goal of MML is to describe medical records, focusing specially on narrative information. The MML standard assures interoperability between systems converting data into MML before it is send. The receiver application then converts it into its own format to use the information. The MML standard describes a large set of types and elements (such as address, telephone number, etc) which are all documented in the specification [20]. As an example it is depicted in figure 2.6 the MML definition of the Telephone Number format.

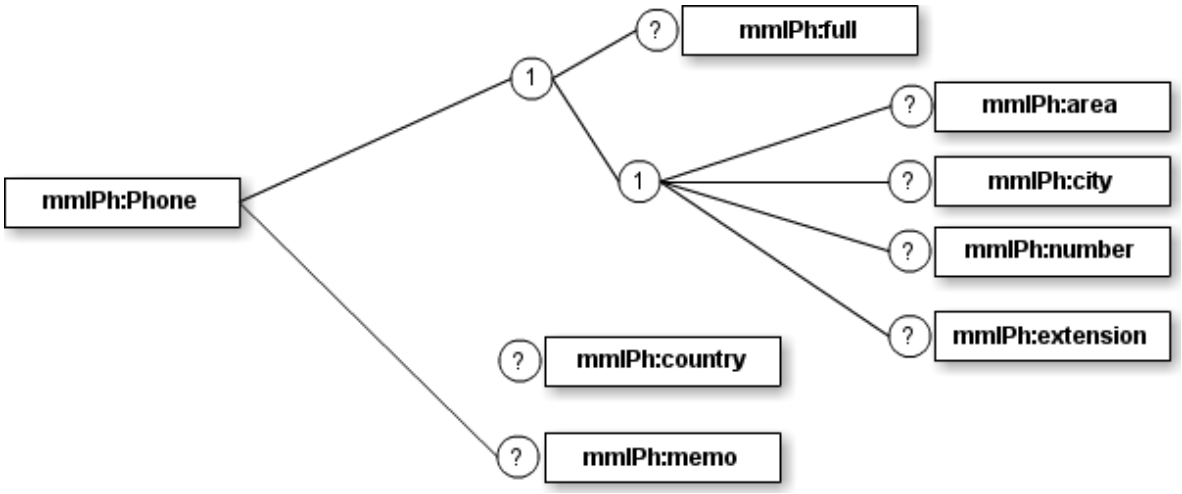

Figure 2.6: EHR Standards - MML definition of Telephone Number adapted from [20]

The header of a MML message (which can be send via HL7 message or any other format) contains many attributes which define an EHR, such as patient demographics, diagnostic information and document author, specifying this way a set of restrictions on the content and structure of a document. There are commercial implementations which support MML, specially in Japan [2].

2.2 Communication Standards

Communication standards define how EHRs should be exchanged between systems. In this group are included WADO, RID and XDS, which are the standards that define access services. These access services consists mostly in operations like query, retrieval and submission of EHR

content. However it is not necessary to support all of this operations, as it is the case of some of the standards described next.

2.2.1 Web Access to DICOM Persistent Objects

WADO is a joint effort of DICOM and International Organization for Standardization (ISO) and published by both organisations [2]. It is not a new medical communication protocol, but a service to view DICOM via web. WADO is a simple approach for accessing particular DICOM objects without requiring the client to speak DICOM. It is a web based service that allows access to and visualize DICOM persistent objects, intending to distribute medical information to healthcare professionals providing a simple mechanism. It offers the possibility to encapsulate the DICOM persistent objects in the HTTP/HTTPS protocol so they can be integrated in Hypertext Markup Language (HTML) pages or XML documents, using the DICOM UIDs(Unique Identifiers for every study, series or image instance) [21]. A query mechanism is not supported. Data may be retrieved in readable form (Joint Photographic Experts Group (JPEG) for instance) or in the DICOM format.

The WADO communication process is a common client/server message exchange. A WADO client issues an Hypertext Transfer Protocol (HTTP) request to a WADO server asking for a specific DICOM instance. The HTTP request is done using the GET command, being the possible values of the query parameter defined in the WADO DICOM specifications [6]. Besides the queries, there are also some options available as, for instance, to request the server to anonymize DICOM objects before transmission or to convert them into a presentation-ready format, such as JPEG. The server than searches for that instance, being capable of transforming it, if necessary, to fit the transfer syntax that was agreed on. Finally the response is again encapsulated in a HTTP message and send to the client [22]. Figure 2.7 shows how to retrieve a DICOM image converted, if possible, in JPEG2000, with annotations added to the image describing the patient name and technical information and mapped into a defined image size.

```
https://aspradio/imageaccess.js?requestType=WADO
&studyUID=1.2.250.1.59.40211.12345678.678910
&seriesUID=1.2.250.1.59.40211.789001276.14556172.67789
&objectUID=1.2.250.1.59.40211.2678810.87991027.899772.2
&contentType=image%2Fjp2;level=1,image%2Fjpeg;q=0.5
&annotation=patient,technique
&columns=400
&rows=300
&region=0.3,0.4,0.5,0.5
&windowCenter=-1000
&windowWidth=2500
```

Figure 2.7: EHR Standards - WADO query to retrieve a region of a DICOM image from [23]

The WADO specification does not define any visualization rules, which means that different server implementations can lead to different rendering results of the same report. This fact can be easily overlooked considering that the WADO service makes it very simple to access DICOM objects, being specially useful for non DICOM clients. WADO is mostly used by web-based EHRs to reference DICOM images or to send them by email. It is also used for

[6] ftp://medical.nema.org/medical/dicom/2009/09_18pu.pdf

making DICOM images accessible through a web browser, allowing image distribution and sharing between distinct EHR systems [2].

2.2.2 IHE Standards

Integrating the Healthcare Enterprise (IHE) [7] is an initiative to improve the way computer systems share information in healthcare and to promote the use of established standards such as HL7 and DICOM. It includes a rigorous testing process for the implementation of this framework and it organizes educational sessions and exhibits at major meetings of medical professionals to demonstrate the benefits of this framework and encourage its adoption by industry and users [24]. IHE has as goal to close the interoperability gaps between the different standards and therefore it provides a process for building a detailed framework for the implementation of standards. The process followed by IHE considers that the care providers are the ones who identify the need for integration, providing manufacturers and information technology professionals a solution for each problem [25]. IHE defined two communication standards: RID and XDS.

IHE Retrieve Information for Display

One of the great challenges concerning EHR systems is the access of patients information across healthcare institute boundaries. RID provides access to persistent objects represented by existing standards (HL7 CDA, PDF or JPEG), as well as to patient specific information, such as allergies, medications or reports, allowing to retrieve and display healthcare documents on systems other than the document keeping system [26]. IHE defined RID as a web service by providing its Web Service Definition Language (WSDL) with a binding to HTTP GET. The RID profile defines two "Actors": Information Source Actor and Display Actor. The Information Source is a database of clinical documents and patient specific information. The Display actor accesses the database to retrieve patient-centric information or persistent documents and to display them to a human observer. The possible transactions between the two actors are also defined by the RID specifications. The communication process is always initiated by the Display Actor and is implemented as a Web Service. There are two possible transactions that a Display may use with the Information Source: Retrieve Document for Display or Retrieve Specific Information for Display. The difference is that while the first is used to retrieve a persistent document, the second is used to access information that is not represented by a persistent document because it is updated more frequently, such as lists of allergies or medications. To retrieve a persistent document the Display provides an Unique Identifier (UID) to identify which document should be send and additional information about the formats supported. The Information Source than sends the document as payload of the HTTP response. When asking for patient specific information the Display provides the patient ID and indicates what kind of information is pretended. The Information Source returns a Extensible Hypertext Markup Language (XHTML) web page as payload of the HTTP response. This page may contain hyperlinks to persistent documents which can be retrieved by using the Retrieve Document for Display transaction.

[7] http://www.ihe.net/

IHE Cross-Enterprise Document Sharing

XDS has the goal to facilitate the sharing of healthcare documents. The basic idea of this integration profile is to store healthcare documents in an Electronic Business using Extensible Markup Language (ebXML) repository and registering them in a document registry in order to facilitate access to each one [27]. It defines two basic concepts: Document Repository and Document Registry. These are two independent units, which serve different functions, being the repository used to store healthcare documents and the registry to keep their meta-data to facilitate the discovery process. The set of healthcare enterprises that agreed to work together on the clinical document sharing process is called XDS affinity domain.

To make the retrieving of a healthcare document as easy and fast as possible, IHE XDS defined a set of actors which communicate with each other using well defined transactions, as shown in figure 2.8. As shown, XDS defines five actors: Patient Identify Source, Doc-

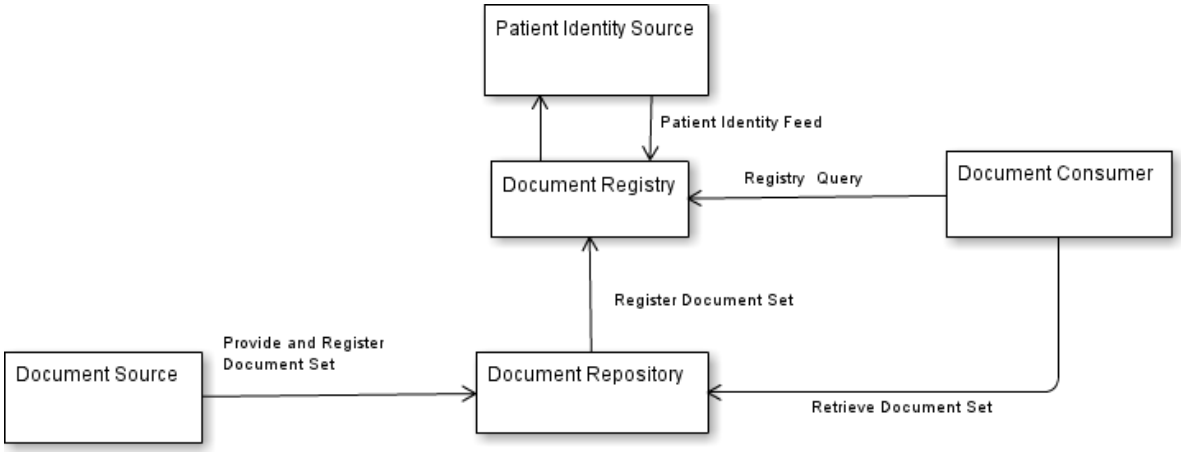

Figure 2.8: EHR Standards - IHE XDS Actors and Transactions

ument Registry, Document Consumer, Document Source and Document Repository. The Patient Document Source sends the document to the Document Repository, using the Provide and Register Document Set transaction. The Document Repository then stores the received document and sends its meta-data to the Document Registry (using the Register Document Set transaction), which maintains the meta-data about each registered document. The Document Consumer actor queries the Document Registry for documents, indicating specific criteria which have to be met (Register Query transaction) and retrieves them from the Document Repository (Retrieve Document Set transaction). The Patient Identify Source provides the patients identification to the Document Registry, using the Patient Identify Feed transaction.

IHE XDS is not concerned with document content, specifying only meta data to locate documents on a faster way. This way the documents may contain any kind of information in any format, such as simple text, structured text (HL7) or images (DICOM). To allow communication there must be agreed on the document format, structure and content [28]. The main functionalities of this standard are the support of documents defining a patient's EHR, the storage of the documents and the possibility to index, query and retrieve them.

2.3 Global Standards

Two of the studied standards define both content and message format, being them EHRcom and DICOM SR.

2.3.1 CEN EN 13606 EHRcom

The European Committee for Standardization (CEN) is a business facilitator in Europe, removing trade barriers for European industry and consumers. Its mission is to foster the European economy in global trading, the welfare of European citizens and the environment. Through its services it provides a platform for the development of European Standards and other technical specifications [29].

The CEN EN 13606 EHRcom is a message-based standard for the exchange of electronic health care records. It does not attempt to specify a complete EHR system as it focuses on the communication between systems and the interfaces needed to ensure interoperability. The first version (ENV 13606) showed many weaknesses as it was implemented, leading its one level modelling approach to a very complex and abstract system. To solve the issues the CEN prEN 13606 EHRcom adopted the OpenEHR archetype methodology to create constrain rules for the information. This way the standard has now five parts [30]:

- reference model
- archetype interchange specification
- reference archetypes and term lists
- security features
- exchange models

So far only the reference model implementation is stable, being the other parts still under development. The Reference Model is a generic model capable of representing the context of an electronic health record of a patient, to support interoperability between systems [31]. The model is divided into several class packages [13]:

- Extract package
- Demographics package
- Terminology package
- Data Type package
- Access Control package
- Message package

The EHR Extract is the top level component of the EHR and defines the data it contains. The EHR content is divided into logical blocks as depicted in figure 2.9, for a better organization of the information. The Demographic package allows the definition of information about persons, organisations and devices that are referenced in the EHR_EXTRACT. The definition of terms used within the EHR are included in the Terminology Package. To define primitive

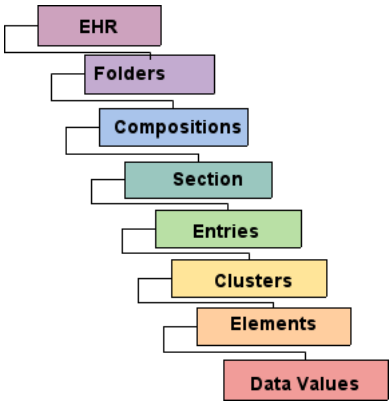

Figure 2.9: EHR Standards - EHRcom Logical building blocks

data types, values for quantities and attributes it is defined a set of data types in the Data Type Package. The Access Control Package defines a representation for EHR access policies. The Message package deals with attributes that will be send via message or serialized form. This package includes HL7 Domain Message Information Model.

2.3.2 DICOM Structured Reporting

DICOM is an international standard defined by American College of Radiology (ACR) and National Electrical Manufacturers Association (NEMA). It defines data formats, storage organisation and communication protocols of digital medical imaging. The DICOM standard is in conformation with the International Standards Organization Reference Model for Network Communications (ISORM) and addresses the issues of conformance. Also it incorporates the concept of object-oriented design [32].

DICOM Structured Reporting (SR) is an extension to DICOM that covers medical reports and other clinical data. It was added to the standard to provide an efficient mechanism for the generation, distribution and management of clinical reports, being mostly used to encode medical reports in a structured manner in a tag-based format. DICOM SR provides a very flexible model to store almost any kind of data (free text to completely structured documents). A structured report has a header information that is also used for DICOM images and the content is represented by a document tree. Parent and child content items are related to each other by a set of relationships [30]. An example of such a tree is depicted in figure 2.10. A structured report is mostly defined by how it is constructed, being capable of containing any kind of structured content. SR documents can be used to represent information like lists, hierarchically structured content, numeric values or references to images. In DICOM SR, each document encodes only the meaning of the information, not how it is intended to be displayed. This means SR documents concern about semantics, not presentation [33]. It is not allowed to reference nodes from another SR report tree, as the content must be fully contained in a single document. If needed, the external information must be copied to the SR document where it will be used.

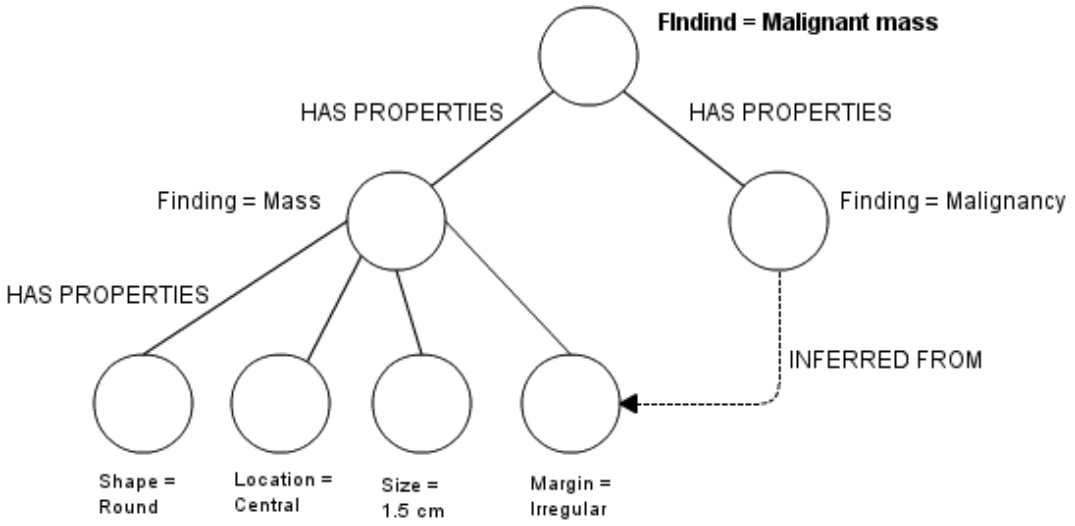

Figure 2.10: EHR Standards - Structured Report tree with references adapted from [33]

2.4 Discussion/Conclusions

The different EHR standards vary in their scope and content. Some specify the content format, some the communication protocol and some define both. The standards EHRcom, OpenEHR, DICOM SR, HL7 CDA and MML provide a definition of the content format. The comparative analysis is shown in table 2.1. All of these standards can be used to store persistent structured documents. The basic unit is the document, however in the case of EHRcom and OpenEHR there is the possibility to aggregate more than one document into a composition. The standards can all contain multimedia objects or references to multimedia objects. All of the content standards follow the two level modelling approach, using to separate models: a reference model and a model to constrain the information. EHRcom and OpenEHR use archetypes to express constraint rules, while DICOM SR, HL7 CDA and MML use templates. DICOM SR is the only standard that has defined a library of possible objects.

Table 2.1: EHR Standards Comparison - Content Format Standards

Functionalities	EHRcom	openEHR	(DICOM SR)	(HL7 CDA)	MML
Store Structured Docs	✓	✓	✓	✓	✓
Document is basic unit	✓	✓	✓	✓	✓
Supports Compositions	✓	✓	✗	✗	✗
Multimedia	✓	✓	✓	✓	✓
Reference to Multimedia	✓	✓	✓	✓	✓
Archetypes/Templates	✓	✓	✓	✓	✓
Defined Library	✗	✗	✓	✗	✗
Specify Distribution Rules	✓	✓	✗	✗	✓

The table 2.2 shows the standards that focus on communication aspects. The basic access services supported by EHR standards are query, retrieve and submission of new documents. Almost all of them support these three services. An exception is WADO which does not

allow querying for document content (need to be combined with another standard) neither submission of new documents. The IHE RID standard does not support the submission service. RID and XDS are content format agnostic, which means that they treat documents as bytes. WADO supports presentation ready formats (JPEG or Graphical Interchange Format (GIF)) or native DICOM objects.

Table 2.2: EHR Standards Comparison - Communication Standards

Functionalities	WADO	EHRcom	(DICOM SR)	(RID)	XSD
Query Service	✗	✓	✓	✓	✓
Submission Service	✗	✓	✓	✗	✓
Retrieval Service	✓	✓	✓	✓	✓
Security	✓	✓	✓	✓	✓
Document Centric	✓	✗	✓	✓	✓
Content Format Agnostic	✓	✗	✗	✓	✓

In conclusion, all of these standards are very similar between them so there is no "best standard". All of them are based on the two level modelling methodology, separating information from knowledge and using archetypes/templates to validate and constrain the information. The main difference between the standards is the progress of the standardization process, so each of these standards can be used to implement an EHR system. The most stable standard is DICOM, however it focuses on medical imaging so it is not suited for a general EHR system. An important point that became evident is that there is no standard which allows full interoperability on its own. For instance DICOM enables interoperability between image departments while HL7 is mostly used for administrative objectives. Other standards provide common vocabulary and semantics. EHR standards also have many ambiguities, therefore different implementations with different results hinder interoperability [22]. The main goal, which is interoperability regarding all scopes, has not been fulfilled yet.

Chapter 3

Planning and Development of an OpenEHR Repository

This chapter describes the adopted solution of an OpenEHR repository. The purpose of the repository is to store electronic health records which follow the OpenEHR standard, offering an API to perform database management functions (like insert, edit or delete a record) and to query the inserted records. To accomplish this task the first step is to do a requisite analysis of such a repository. There are a few aspects that have to be considered. A very important one is the fact that the repository must store OpenEHR records which have a very specific need: a dynamic database model. As the OpenEHR records are instances of a reference model and constraint by a separate level (archetypes) it is not possible to define a constant database model. The final implementation should be able to store any record which follows the structure specified by the OpenEHR standard. Figure 3.1 shows the OpenEHR architecture, which has been explained before. Here it is used to explain where the repository to implement will be inserted. Following the two level modelling approach it results this generic model where information is separated from knowledge. The reference model supports the record management functions and ensures sharing with other providers. The second level (archetype model) models the knowledge using archetypes. This way archetypes are created by domain experts (doctors in this case) and used to validate the information inserted by users. As shown in the figure, the repository will store reference model instances which are constrained by the archetype model. The advantages of this model is its generic nature. The medical area is always changing, so it is important to be prepared to accept new information structures in the repository. This way everything can be stored, as long as there is an archetype which validates the data. So the objective is to implement a database capable of storing OpenEHR reference instances which are validated by the archetype model. Another fact to have in consideration it that the repository is only a part of a healthcare system and consequently has to be inserted into the architecture of such a system. To make this possible the implementation should have a well defined set of services which are reusable by any application. This services should provide the functionalities normally supported by an information system, being them the management of records (insertion, update and removal of records) and the execution of queries against the inserted information. Finally should be implemented a web interface to demonstrate a possible usage of an OpenEHR repository and to make the implemented functionalities "visible".

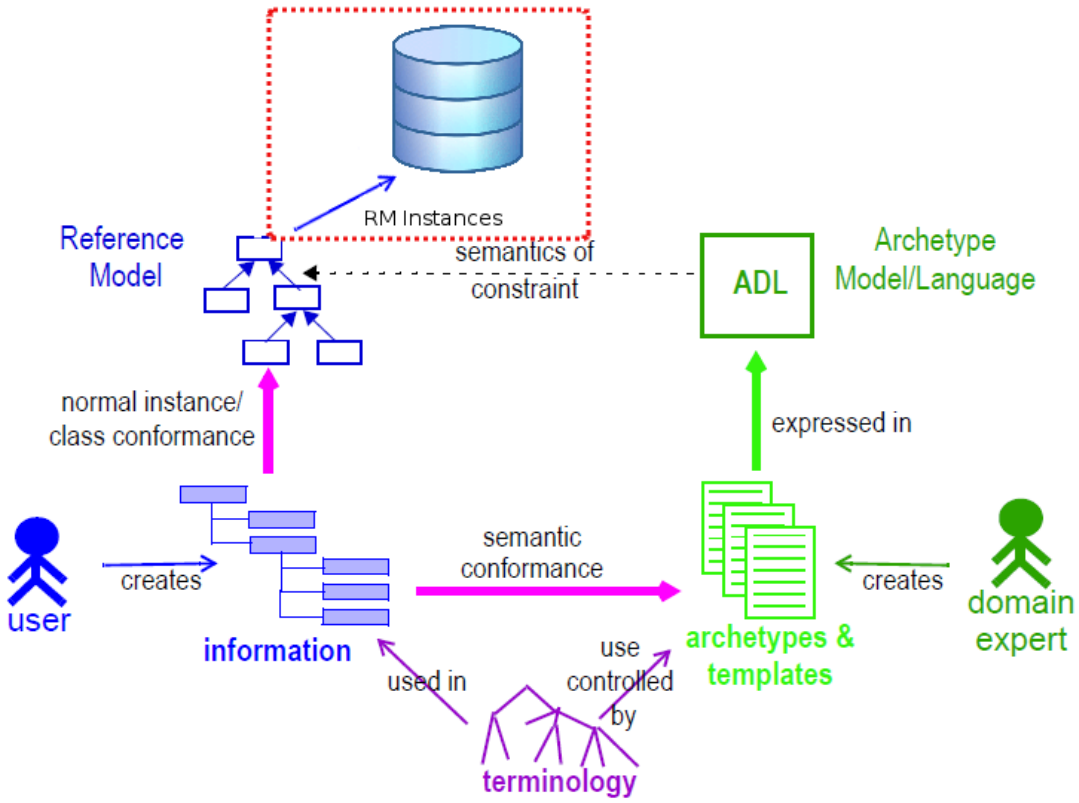

Figure 3.1: OpenEHR Repository - Scope within the OpenEHR Architecture

3.1 Architecture

The system architecture was designed dividing the desired functionalities into different layers. The result is shown in figure 3.2. This section gives an overview on how the repository is structured, being the used technologies explored in detail in the following sections. The Core Layer consists in the BaseX repository, where the OpenEHR records are stored. The Service Layer is the layer responsible for offering the services that allow the manipulation of the records inserted into the repository. The last layer (Application Layer) consist in user applications which make use of the offered services. Communication between layers is done via Web Service (Representational State Transfer (REST) and Simple Object Access Protocol (SOAP)). The system follows the Service Oriented Architecture (SOA) model, which means a service oriented architecture [34]. To introduce the SOA approach it is important to know that the Internet has evolved much since its start in the 60's. Nowadays connectivity drives the emergence and convergence of technologies. So today the value is not defined

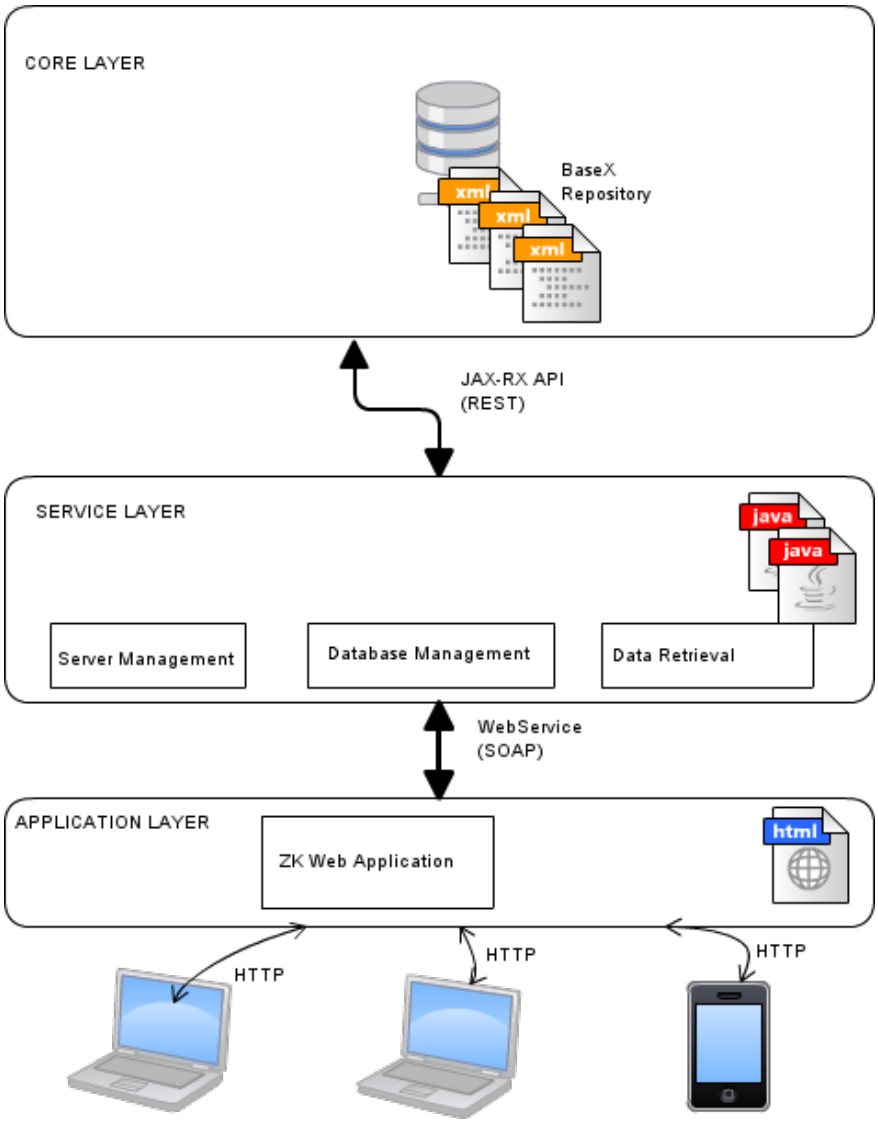

Figure 3.2: OpenEHR Repository - System Architecture

as much by functionality but by connectivity. The new programming model that follows that philosophy is called Service Orientation. A service oriented architecture is a group of services that communicate with each other. The approach consists in developing individual, independent services with an own and deterministic functionality. These services can be re-utilized. To form an application this services are connected with each other performing each one of them a specific part of the whole functionality. Service Orientation presents the ideal that functionalities are distributed clearly and independently. So each service can function without the others and be reused by whichever application needs it. It is important that services have a well defined interface and can be discovered and accessed. Because the functionalities of an application in this model are spread, a communication way is needed. One very commonly used are Web-Services.

A Web-Service is a component with a specific functionality that can be accessed via stan-

dard Internet protocols. A Web-Service functionality can be used without concerns about how it was implemented, so they need to have a well defined interface explaining which functionality is supported and what parameters must be passed. There are a few technologies and specifications which define how a Web-Service should be constructed and used. These standards concern about a common way to define data, a common message format for information exchange, a language to describe the supported functionalities and a mechanism to localize the services. XML is a natural choice for data representation and therefore it is used by many specifications to define how data is represented with a XML Schema to describe data types. SOAP [35] is a light protocol for data exchange. It not only includes a rule set on how to use XML to describe the information but also describes the message format, conventions to represent Remote Procedure Call (RPC)s using SOAP and HTML associations. It is mostly used for integration in enterprise applications. An alternative is REST which considers that each Uniform Resource Loader (URL) is the representation of an object. HTTP methods such as GET, POST, PUT or DELETE are used to query/manipulate this objects. WSDL [36] is a XML based language used to document the messages the Web-Service accepts and generates. There is also needed a way to know what services there are and what they do. The Disco protocol (Discovery Protocol) defines the format for the document discovery and a protocol to retrieve the document, allowing the discover of services on a known web site. However normally this web site is not known. The UDDI (Universal Description, Discovery, and Integration) is a mechanism where the services can be announced and localized. To conclude, a XML Web-Service can be defined as a software service published on the Web using SOAP/REST, described in WSDL and registered in a UDDI.

3.2 Storage of OpenEHR records

The Core Layer is responsible for managing the information to be stored. It is this layer that contains the repository which stores the EHR records. The storage of health information is a difficult topic, due to the always changing nature of the information to store. There is constantly made progress in the medicine field and consequently there are often new data types that need to be stored and retrieved. OpenEHR presents a two level modelling approach, which creates a separation between the information model and the rules to constrain this information. This makes it necessary to find a very flexible storage model that achieves to follow the progress and change of the records. To design such a repository were studied some of the existing storage technologies which will be described in the next section.

3.2.1 Storage Technologies

This section describes the storage technologies that were considered for the repository implementation, being them relational databases, object oriented databases, XML databases and plain text storage.

Relational Database

Relational Database is the most common used and known database model. All data is stored in tables logical connected through relations, being a table a collection of data of the same type. Any table has to share at least one field with another one to establish their relationship (which can be one to one, one to many or many to many) [37]. This approach

is very effective if the application deals with data that is easy to convert into this kind of model. For many applications this it not the case, as the data consist in a set of attributes (object) and has to be divided in its parts to be stored and reassembled if queried. For this kind of data it would be more appropriate to use an object-oriented model To query data stored in a relational database it is used Structured Query Language (SQL). SQL is a language designed to manage relational data. It is the most widely used query language as it allows operations upon the data which include insert, update, delete and query. The most common operation in SQL is the query, as it is possible to select data from one or more tables or according to expressions with the SELECT instruction. There are both proprietary relational databases such as Oracle, Microsoft and IBM and open source implementations such as MySQL, PostgreSQL and SQLite.

Object Oriented Database

Object Oriented Databases are characterized by storing objects rather than simple data as strings or integers. An object consists of attributes that are used to define it and methods which define the behaviour of the object [38]. Objects are used in object oriented languages such as Java and C++. This kind of database model should be used always when data is complex or has complex relationships, as with object oriented databases there is no need to assemble and disassemble objects. They also feature easier navigation and better concurrency control than relational databases [39]. For applications developers it often makes it easier to model this kind of database in comparison with the relational model, since they model data as it is in the real world without having to normalize the information into different relation tables. For simple data with simple relationships it is although recommended to use a relational model as it shows more efficiency and simpler tables. There is no standard query language for an object oriented database, however Object Data Management Group (ODMG) has defined the Object Query Language (OQL) [40] which is very similar to SQL and supported by some vendors. It is also possible to use one of the proprietary languages instead.

XML Database

A XML Database is a storage software that allows to store data in XML format. This data can than be queried using different kinds of technologies which will be presented further ahead. The use of the XML to represent information has increased over the years. The first and more obvious advantage is the fact that it allows to write customised tags, as there is no restriction to the tags offered by proprietary vendors (such as it happens with HTML). This allows a very flexible data model and makes it easy to add/remove new data fields. More important than defining tags is the fact that there is also the possibility to define the rules to constrain the data this tags will contain. This makes the data extremely portable, since it comes with a definition that describes how it should be used. XML is also platform independent, as it is simple to understand and easy to read for both humans and computers.

There are two types of XML databases to consider: XML enabled databases and native XML databases. XML enabled databases use the relational method as storage model. The XML data is stored in a Character Large Object (CLOB) column. Some technologies, as for instance Oracle [1], allow also to register a Schema for validation and store the information

[1] http://www.oracle.com

in a schema based XML type using Object-Relational storage. The second alternative is a native XML database. Here the XML documents are directly indexed and the entire XML document and related elements are stored. The advantage of this solution is that it keeps all the content in one place so it easily queried. The queries are also very fast as there can be used indexing of the content as it fits best regarding the application under development.

Associated to XML databases are a few technologies that must be referenced. Before explaining the query technologies there is another very important technology related to XML: XML Schema Definition (XSD) [41]. A XML Schema describes the structure of a XML document, being a XML alternative to DTD (Data Type Definition). It defines the legal building blocks of a XML document, constraining for instance which elements can appear or the order of child elements.

To query XML data that has been stored it is used XQuery [42]. XQuery is build of XPath expressions and is to XML what SQL is to relational database tables. The main point is that is allows to query data that is contained by multiple sources. The basic building block of XQuery is the expression which is a string of unicode characters. XQuery provides different kinds of expressions which may be constructed from keywords, symbols and operands (can also be other expressions). Another important step was the development of Extensible Stylesheet Language (XSL) [43]. It was designed for the purpose of describing how the information should be displayed. Besides fulfilling what it was designed for it was also the start of other important technologies such as XPath and Extensible Stylesheet Language Transformations (XSLT). XSLT [44] features the transformation of XML documents into other formats. The transformation describes the rules that should be used to transform a source tree into a result tree. This is done based on templates that are matched against elements of the source tree. XPath [45] is used to navigate through elements and attributes in a XML document. To accomplish that it uses a path notion such as URLs for navigation through the hierarchical structure of a XML document. It also provides functionalities for manipulation of strings, numbers and booleans.

There are many databases that support the XML data type so here will just be referenced the most important ones having in sight the requisites of the application to develop: eXist [2], BaseX [3] and DB2 Express-C [4]. eXist and BaseX are both open source database management software, written in Java, ready to store and query XML documents. They offer a RESTful and XML:DB API BaseX has the plus that is provides the XQJ API (XQuery API for Java) [46]. DB2 Express-C is a full functional relational and XML data server from IBM. There is a free community edition of the DB2 data server. Since version 9 it supports XML native storage. This is done by defining a new data type XML that can be stored in the table columns. The documents are stored in parsed hierarchical format. For querying the data there can be used both Xquery and SQL with XML extensions. It also differs from the other ones being ideally suited for use in cloud environments.

Plain Text

Plain text database systems store data in a common text or binary file. The file contains usually one record per line and data types and attributes are defined by convention and there is no structural relation between the records. To retrieve the information the files must

[2]http://exist.sourceforge.net
[3]http://basex.org
[4]http://www-01.ibm.com/software/data/db2/express/

be parsed. There is also the possibility to index (using for instance Apache Lucene [5]) the information so there is a much shorter answer time for each query. This approach is best suited when the data is simple and has no complex relations as it would be very difficult to process complex joins. This option was considered for implementation of a file system based storage with indexation.

Storage Technologies Assessment

There is no best database. It is necessary to analyse the nature of the data to store and the functionalities to support by the applications to decide which database model fits best our needs. For instance non-tabular data cannot be saved in tables without loss of performance and flexibility while naturally a object oriented database would fit best. XML databases are indicated when the data model is very flexible and must be easily modified.

The choice of which one to use was based on the needs of the repository, being the chosen one Native XML Database. First of all, native XML databases is a relatively new technology, so there are no OpenEHR repositories which use it. This makes this a interesting case of study, to compare performance with relational approaches or with no native XML databases. No native XML databases use a relational model and have XML fields to store the data. XML is optimal for document centric applications. This is perfect for an EHR repository, as the information is patient centric, being it possible to have a XML file per patient which contains all of his medical data. Another advantage is that XML is a known standard so it facilitates the communication with other systems, increasing interoperability. Within the existing software to manage native XML databases were BaseX considered the best fit regarding the repository's needs and goals.

3.2.2 BaseX Overview

The database system chosen to implement the repository is BaseX. BaseX is a light-weight, high-performance, scalable XML database system and Xpath/XQuery processor, including for support for the Worl Wide Web Consortium (W3C) Update and Full Text extensions [6]. The software is open-source, completely written in Java and platform independent. The update functionality is a great advantage regarding performance. This because it allows to access a specific position of the file and to modify its value. What normally would have to be done is to read the file, change it and rewrite it. So it brings definitely a speed increase. Its client-server architecture makes it suitable for a system with distributed database access. It offers a RESTful API for accessing distributed XML Resources. REST facilitates a simple and fast access to databases though HTTP (using HTTP methods GET, PUT and DELETE). The REST implementation is based on JAX-RX, an interface layer to provide unified access to XML databases and resources [47]. JAX-RX uses the HTTP server Jetty. Jetty provides a HTTP server, HTTP client and javax.servlet. The javax.servlet packet contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment [7].

The baseX source code is available for free (read-only). The most important packages can be visualized in figure 3.3. The **basex** package is the main package of the project. It

[5] http://lucene.apache.org
[6] http://basex.org/
[7] http://download.oracle.com/javaee/5/api/javax/servlet/package-summary.html

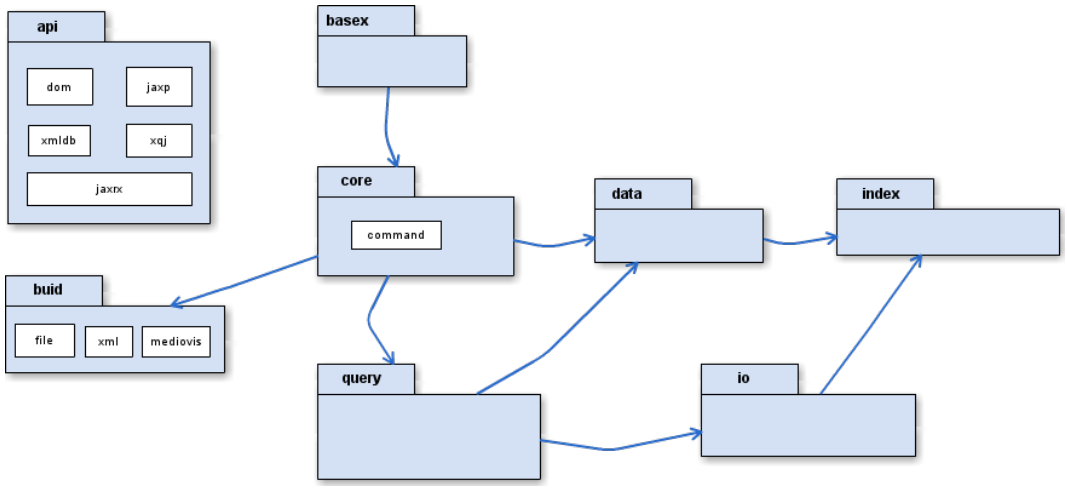

Figure 3.3: BaseX overview - BaseX package diagram

contains all the starter classes with main methods (stand-alone console mode, client console mode, graphical front end and database server). The database core classes are contained in the **core** package. It provides the implementation of all available commands (add, create, find, etc) but also classes to represent users or manage notification triggers. The **build** package contains classes for creating new database instances. This classes have three different implementations available: file (for creating databases from different sources), XML (for creating databases from XML documents) and mediovis (for creating databases from MAB2 library data). The query package contains the XQuery implementation. This includes a QueryProcessor, a XPathProcessor and a XQueryProcessor. The **io** package contains Input and Output classes. This classes allow reading and writing of data to data structures, grant access to databases files, allows to store tables on disk and read it block-wise and enables main memory access to a database table representation. The database index structures can be found in the **index** package. BaseX allows to index names as well as values. Additionally to the common indexes, BaseX allows full-text indexation. Along with the index structures there is also a scoring method implemented. The **data** package contains the classes that define the database structure. The implemented model can be divided into four main classes (view figure 3.4): Data (provides access to the database storage), DiskData (stores and organizes the database table and index structures for textual content in a compressed disk structure), MemData (stores and organizes the database table and index structures for textual content in a compressed memory structure) and MetaData (provides meta information on a database).

The last package (**api**) contains the API implementations BaseX offers: Basic Document Object Model (DOM) API, Basic JAXP API, implementation of the JAX-RX API, implementation of the XML:DB API and implementation of the XQuery for Java (XQJ) API. All this classes are used to perform the common database management functions. For instance when a new database is created, the command is stored in *Command* and a process is created (instance of *Proc*, which can be found in the core package). The process is run and a *XMLParser* is created . The database is build by the *DiskBuilder* and a instance of *DiskData* is returned. Indexes are added by *IndexBuilder* instances, creating *Index* instances and data reference is registered in *Context*. This process takes approximately 32 ms.

The response time of a query is the sum of the time it takes to perform four tasks: parsing,

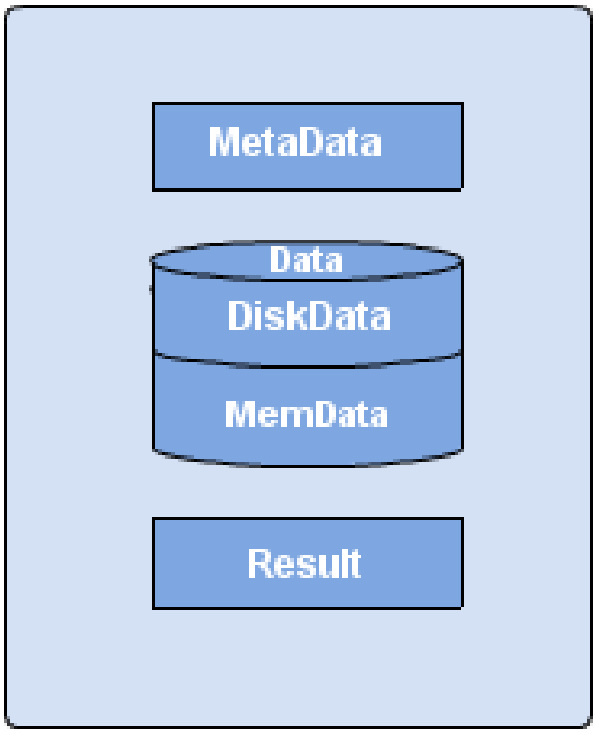

Figure 3.4: BaseX overview - BaseX data package

compiling, evaluating and printing. The evaluating process is which takes the most part of the time (99.7%), parsing takes 0.17% and compiling and printing divide the rest. Obviously the printing depends on the number of result that are devolved. There are three strategies that can be used for the evaluation process: sequential database scan, full text index based evaluation and hybrid. The process is chosen by the compiler according to what he considers more effective.

BaseX Storage

BaseX uses the Pre/Post encoding storage model [48], which has registered great performance advantages in relation to other XML storage models. This model stores a pre/dist/size combination for each node, being the size attribute mainly used to accelerate child and descendant traversals, and the dist attribute to allow access to the parents and ancestors of a node by saving the distance of the node to its parent. Response time of index-based queries can be reduced if a fast access to ancestor nodes is possible. The flat storage of XML documents has the advantage to allow sequentially parsing of the documents, allowing a very fast query specially when sub-sequent nodes must be accessed. Another great advantage of the BaseX storage methodology is that the final table contains fixed-size entries, since attribute names are indexed and texts and attributes are stored separately. This leads to no variable-sized tuples, allowing a easy calculus of memory/disk offset of XML nodes [49]. Additionally, at the end the node tuples are compacted which leads to a further speed up of node access by minimizing the tuples size [50]. To demonstrate how data is saved by BaseX is used the XML example of figure 3.5. It shows a very simple extract of person data stored in XML format, consisting in one document (named students), three XML elements (person, name and age),

```
<person id="1">
   <name>Linda Velte</name>
   <age>22</age>
</person>
```

Figure 3.5: BaseX storage - XML example

one attribute (id) and two text values. This distinction between the elements is important for the decomposition of the document into bytes to store, keeping the structure and relations of the data. The first step consists in storing the document in a simple table (figure 3.1) with the following attributes:

- **PRE** internal integer reference to an XML node
- **PAR** parent PRE value
- **SIZ** number of descendant rows
- **ATS** number of attributes
- **KND** node kind (element, attribute, text, document)

Table 3.1: BaseX storage - Simple Table example

PRE	PAR	SIZ	ATS	KND	CONTENT
0	-	6	-	DOC	students
1	0	5	1	ELEM	person
2	1	-	-	ATTR	Id="1"
3	1	1	0	ELEM	name
4	3	-	-	TEXT	Linda Velte
5	1	1	0	ELEM	age
6	5	-	-	TEXT	22

This table is then transformed as shown in figure 3.2. As PRE value is dense it does not need to be stored. PAR is converted to DIS (distance to parent). SIZ and ATS are increased by 1 (pointer to next XML sibling). The Content is mapped to references: TAG (XML node tag), ATN (attribute name), ATV (attribute value) and TXT (text). This last table is then converted to hexadecimal and written to BaseX's tbl.basex file. The table is compressed, removing holes. For convenience regarding the storing to disk process, there are written 16 bytes per row, being the last 4 bytes used for an unique node ID. The maximum number of nodes is 2^32 (4GB) and text limit is 2^39 (500GB). This limitations are no concern in the perspective of the OpenEHR repository. For performance evaluation purposes were inserted 30000 files into the database (corresponds to 30000 patients), which occupied only 85 Mbs.

Table 3.2: BaseX storage - Simple Table Transformation example

DIS	SIZ	ATS	KND	TAG	ATN	ATV	TXT
-	7	-	0	1	-	-	-
1	6	2	1	2	-	-	-
1	-	-	3	-	1	0	-
2	2	1	1	3	-	-	-
1	-	-	2	-	-	-	0
4	2	1	1	4	-	-	-
1	-	-	2	-	-	-	1

BaseX Indexation

BaseX supports four kinds of indexes [8]. This indexes are used to speed up text comparison in predicates (text index), attribute comparison in predicates (attribute index), full-text queries (full-text index) and resolution of location paths (path summary). Text and attribute indexes are based on a balanced B-Tree (generalization of a binary search tree in which a node can have more than two children) and support match and range queries. The full-text index is implemented as a sorted array structure and is used for simple and fuzzy searches. The full-text index supports a number of options to optimize execution for a variety of scenarios. The options included are:

- **Language** BaseX comes with German and English but other languages can be added using stemmers from Lucene [9] or Snowball [10].
- **Stemming** Tokens are stemmed with the Porter Stemmer [51]. In a stemmed index queries are made of the stems of the words. For example when searching for the token *information*, the search will return this term but also the ones containing *informations* or *informed* because the common stem is *inform*.
- **Suppports Wildcards** wildcards can facilitate the search allowing the use of characters to replace some information in the search. For example the query *l?ve* will return records with the word *live* as well as with the word *love*. The efficiency of wildcards is lowered when stemming is used. Returning to the previous example where the stem is *inform* (from the token *information*), *information?* will no longer return results because the index no longer contains the whole word.
- **Case Sensitive** Tokens are index in case sensitive mode
- **Diacritics** Diacritics are indexed as well
- **TF/IDF Scoring** Term frequency/inverse document frequency is a weight used to define how important a word is to a document. This importance is calculated having in consideration how many times the word appears in the document and how many times it appears in the corpus (words that appear often in the corpus are not that

[8] http://docs.basex.org/wiki/Indexes
[9] http://lucene.apache.org
[10] http://snowball.tartarus.org

important to a specific document). BaseX offers three scoring types: one standard that considers the length of a term and its frequency in single text node, standard TF/IDF algorithm which treats document nodes as document units and one last one, which is recommendable for XML files which contain only one document node, as each text node is treated as a document unit in the TF/IDF algorithm.

- **Stopword List** A stop word list can be defined which has the purpose to indicate which words are not useful to index (such as the tokens *the*, *or*, *an*, etc).

3.2.3 Database Structure

The repository consists in one unique XML node which contains the EHR records. There is one OpenEHR record per patient. The OpenEHR EHR's structure can be observed in the following figure 3.6. Each object is identified by an EHR id and contains structured, versioned information, plus a list of Contribution objects that act as audits for changes made to the EHR. The EHR Access object contains the access control settings for the record. The EHR

Figure 3.6: Database Structure - High Level Structure of the OpenEHR EHR from [52]

Status object contains status and control information such as if the record is queryable or modificable. Optionally it can include the patient id. This id may be suppressed when the relation between patients and respective EHR id is stored separately. The Directory object is a Folder structure that can be used to hierarchically organise Compositions. Compositions, EHR Status and EHR Access objects are versioned. Contributions are used to save every change made to the record. Every commit causes a set of versions to be committed in one go. Rolling back means retrieving the data at each Contribution point, not just arbitrary points in time. A contribution contains information about the changes made, who did the changes and at what time. The composition object is the object that contains the medical information of the patient. It follows a well defined structure defined by the OpenEHR standard. This structure can be observed in figure 3.7.

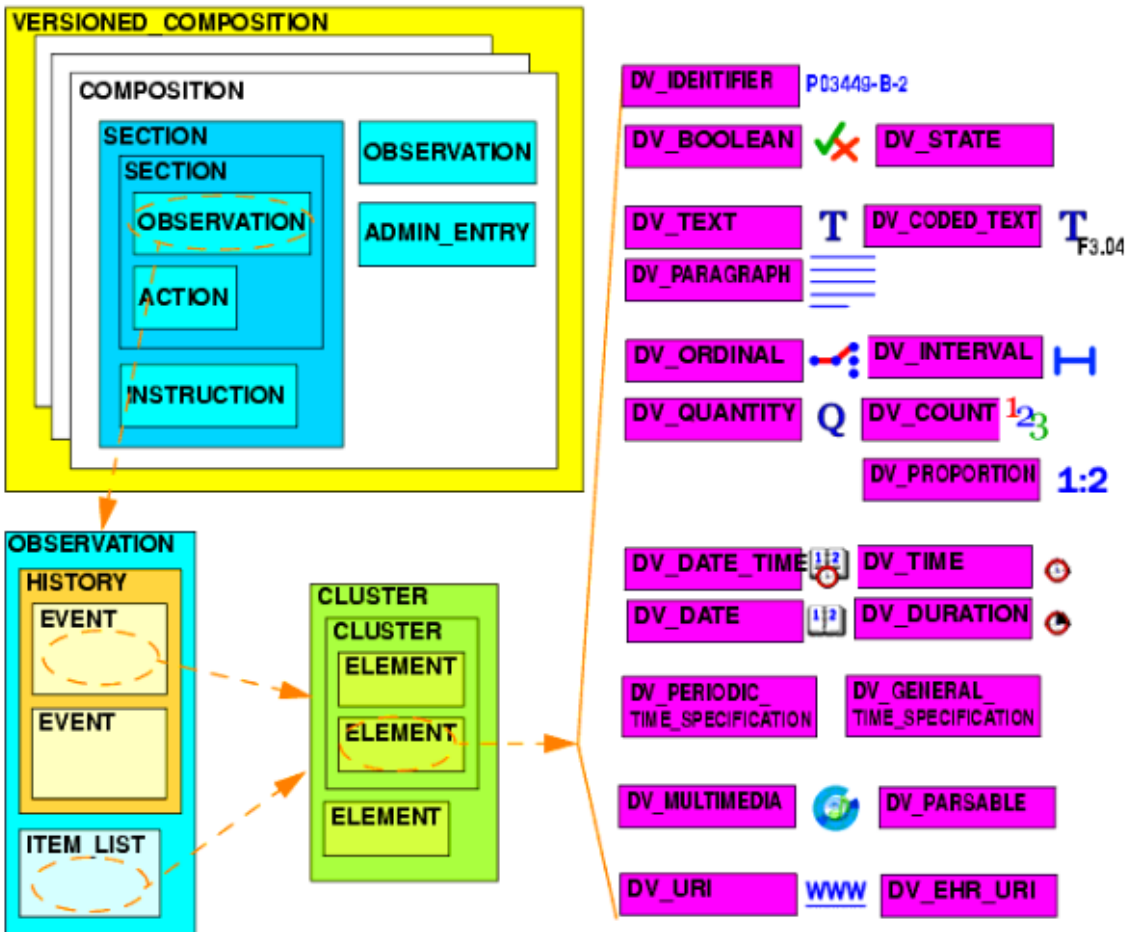

Figure 3.7: Database Structure - Composition Structure from [52]

All clinical information within a OpenEHR record is expressed in "Entries", which are single clinical statements and intended to be archetyped. There are five subtypes of the Entry class: ADMIN_ENTRY, OBSERVATION, EVALUATION, INSTRUCTION and ACTION. These subtypes allow to represent the interaction between patient and clinical investigator systems. A problem is solved by making observations, forming opinions and prescribing actions (descriptions) and, finally, executing the instructions (actions). Each of this classes has several subtypes to make it possible to medical professionals to represent the data (they can not think in terms of only five kinds of data). So, for instance, a instruction can be a investigation request or a intervention request. This way every clinical information, from initial observations to medication list can be stored in a compositions instance, in spite of being completely different kinds of information. This allows to have a patients whole medical history in a single record without making any further effort.

The records inserted into the repository are reference model instances. To create them was used the Java Reference Model Implementation from Opereffa [11]. Figure 3.8 shows the package diagram of the OpenEHR Reference Model. The implementation uses the Java 5.0 platform and uses several open source libraries, such as log4j and commons-lang from the

[11] http://www.openEHR.org/projects/java.html

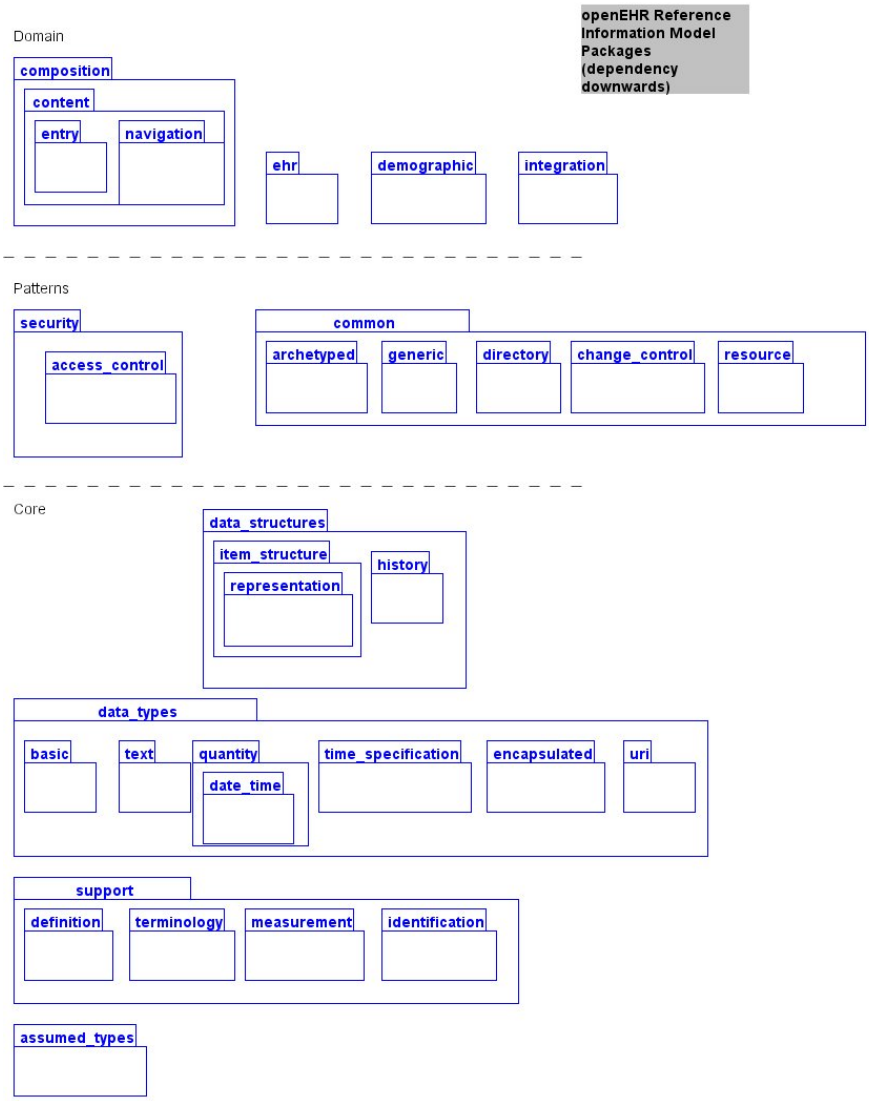

Figure 3.8: Database Structure - OpenEHR reference model package diagram.

Apache Software foundation [12]. The goal of this implementation is to keep the Java look and feel and simultaneously be faithful to the OpenEHR specification. To assure interoperability with applications implemented in other programming languages it is very important to use the Java language correctly when mapping between OpenEHR assumed data types and native Java types. The Java implementation is released under three open source licences: General Public License (GPL), Lesser General Public License (LGPL) and Mozilla Public License (MPL), having the user the possibility to choose which one fits best into his application purposes. The build is managed by Maven from the Apache Software Foundation, which provides comprehensive support of software projects and facilitates the dealing with dependency libraries. The implementation results in six main components: openEHR-rm, openEHR-aom, openEHR-ap, adl-parser, adl-serializer and rm-builder. The openEHR-rm is

[12] http://logging.apache.org

the base component that provides the Java implementation of all OpenEHR Reference Information Models (it implements the OpenEHR RIM, which includes the classes Common, Support, Data Types, Data Structures, Demographics and EHR). The openEHR-aom implements the OpenEHR archetype model, providing object validation and construction. The openEHR archetype profil is implemented by openEHR-ap, which provides implementation of the domain data types. The OpenEHR Archetype Definition Language is implemented by adl-parser (translates ADL to AOM) and by adl-serializer (serializes AOM to ADL). Finally, the rm-builder, which is the Reference Model objects builder, implements the OpenEHR semantics specification.

To create OpenEHR records for testing purposes was created a Java class which is used to create reference model instances and fill them with random data. Each record is initially created with ehrAccess (uid, archetypeNodeId, name, archetypeDetails, settings), ehrID (value, root), ehrStatus (uid, archetypeNodeId, name, archetypeDetails, subject, isQueryable, isModificable), systemID (value, root) and timeCreated (dateTime, value, fractionalSecKnown, isPartial, minuteKnown, secondKnown). Later, when a user needs to add a composition, it is added a versionComposition tag which will contain all further inserted composition instances. Every time a change is made to the record, is also created a contribution instance, stored in the versionedContribution tag, to reflect the changes made and register control information (author and date of the alteration). Besides the class to create the reference model instances was needed a auxiliary class to make the conversion from Java object to a XML String. To do this were used the classes from the javax.xml.bind package [13], which provides a runtime binding framework for client applications including unmarshalling, marshalling and validation capabilities. Using the classes JAXBContext and Marshaller it is possible to convert a class instance into a XML String in one simple step. The only thing that is needed are the right XML annotations at the attributes of the class to marshal.

Composition Validation

The key information in a composition can be found in its content, composer, and context attributes, as showed previously. It is not important to explain all these fields, since what matters is that all archetypes (which define the compositions, present and future ones) follow this composition structure. This said, it is reasonable to validate them using XSD files which verify if they follow the given structure. This is done to every composition before it is inserted into the repository. Figure 3.9 shows the XSD schema to validate compositions to be inserted. Each element displayed has its own XSD validation file, allowing to verify if a XML file follows strictly the structure dictated by OpenEHR. To validate a XML file using the created XSD is used the javax.xml.validation API [14], which uses three classes to validate a XML document: Schema, SchemaFactory and Validator. A Schema instance is created using a SchemaFactory instance and indicating the path of the XSD file to use. The validation is then done with a Validator instance comparing the XML structure with the created Schema.

Version Control

It is assumed that an EHR (with EHRAccess, EHRStatus and Composition objects) will be modified, being it possible to add new created objects or edit existing ones. Examples of

[13] http://download.oracle.com/javase/6/docs/api/javax/xml/bind/package-summary.html
[14] http://download.oracle.com/javase/1.5.0/docs/api/javax/xml/validation/package-summary.html

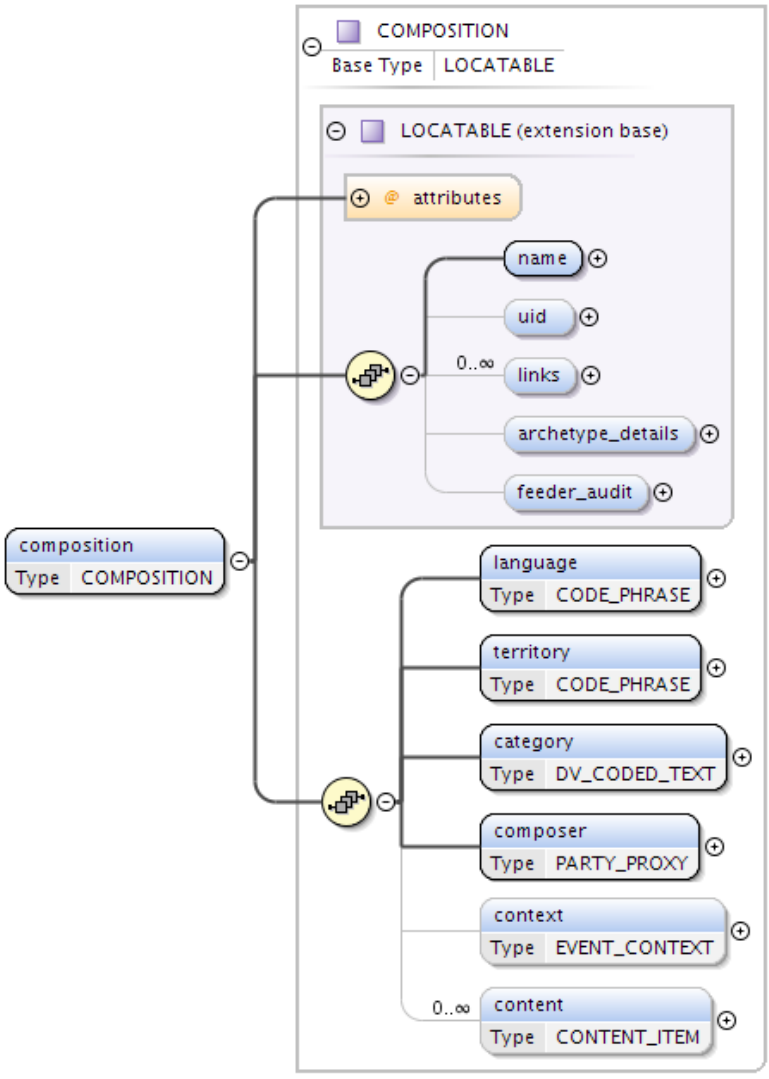

Figure 3.9: Database Structure - Composition XSD

updates are the addition of a new composition when a patient had a new test performed and the clinical professional registers the results or the modification of an existing composition when a new medication is added to the medication list. No matter what changes are made there are three aspects that must always be met [53]:

- the record should always be in a consistent informational state
- all changes to the record must be audit-trailed
- all previous states of the record must be available for the purposes of medico-legal investigations

To accomplish that OpenEHR defines (in its Common Information Model) change control and versioning facilities, which are based on the existence of change-sets, called Contributions. A contribution is created every time there is made a modification to the record, consisting this modifications in every update from composition addition to error corrections or changes due to software upgrades. The list of contributions is attached to the EHR along with the changes made to the data, making it possible not only to capture top-level object changes but also to know exactly which change set resulted in which user commit. The contribution object defined by the OpenEHR standard contains an ID, audit details (information about the person who did the changes and when there were made) and data about the changes specifically (kind of update and earlier versions).

3.2.4 Repository result

The implemented repository follows the structure dictated by the OpenEHR standard. Each EHR contains an EhrAccess(uid, archetypeNodeId, name, archetypeDetails, settings), EhrStatus(uid, archetypeNodeId, name, archetypeDetails, subject, isQueryable, isModifiable), EhrID object(value, root), systemID (value, root) and timeCreated (dateTime, value, fractionalSecKnown, isPartial, minuteKnown, secondKnown), followed by the list of versioned compositions and contributions. All this information is versioned, which means that it is possible not only to go back to any older state but also to reconstruct who made which changes to the record. This fact is very important as in the medical area a history of a patients health information has to be kept for further possible legal investigations.

The result of the repository can be observed in figure 3.10, which represents the database structure viewed in the BaseX GUI (which allows an interactive visualization of the XML data, as well as the execution of queries using XPath and XQuery). To insert a record it must be created an OpenEHR record in XML format containing EHRAccess(, EHRStatus and EHRID objects. It may also contain compositions inside the versionedCompositions tag, being it also possible to add them later. Additionally the versionedContributions tag contains the list of contribution objects, reflecting the change set made to the record since its creation. The fact that the database consists in one single XML file, having each patient a root node with all his medical information (one OpenEHR EHR) makes this repository optimal for patient centric queries. Information centric queries are also possible, as the XML language has a very flexible query language (XPath) which allows to query for attributes and values.

Figure 3.10: Database Structure - OpenEHR record structure

3.3 Service Layer

The service layer contains the services that make the connection between the repository and the user interface allowing the manipulation of the state of the database and the query of the inserted information. It is constructed in a way that this services can be reused by any application, being the created web interface (which uses them to access the repository) just for demonstrative means. Following a service oriented architecture, a service consists in a function that is well-defined, self-contained and does not depend on the context or state of other services. It is a discoverable software resource which has a service description available for searching, binding and invocation by a service consumer [54].

The set of services implemented cover the stipulated requisites which are allowing the management of electronic health records storage and the query of information contained by these records. To connect to the XML repository it is used the JAX-RX API, offered by BaseX [15].

3.3.1 JAX-RX

REST is an architectural style of networked systems, based on the idea that each request is encapsulated and valid by itself and each resource is directly accessed. While REST is not a standard it makes use of standards, such as HTTP, URL and XML [55]. To make use of this paradigm within Java environments, JAX-RS was introduced, which is a Java API for RESTful Web-Services. The JAX-RS specification defines a set of Java APIs for the development of Web-Services built according to the REST architectural style. It simplifies and unifies the task of parsing the URL, which contains the request, and offers a way to access resources.

Since XML is a core technology for web-based applications, the Distributed Systems Laboratory and Database & Information Systems Group from University of Konstanz in Germany, introduced JAX-RX (Java API for RESTful XML resources). This approach enables third party applications to be accessible over a platform independent, up-to-date interface layer [56]. An important aspect of JAX-RX is that all URLs have the same layout, containing information about the server, the implementation, the resource which wants to be accessed and the list of parameters. The different functionalities are represented by the known REST methods (POST, GET, DELETE and PUT) and for each of them are defined some extensions. The GET and POST methods allow users to directly specify parameters in the target URL. The list of optional parameters includes **query** (XPath/XQuery query expression), **xsl** (XSLT transformations), **output** (XSLT/XQuery serialization options), **count** (number of items to be returned, **start** (index position of the return values), **wrap** (resulting wrapping by additional parameters), **revision** (revision of the data) and **command** (implementation-dependent database command). A DELETE request allows to remove a source (a database, a collection, a resource within a database or a directly accessible node). As no additional parameters are allowed by the DELETE request, only those resources which can be referenced by a path can be removed. A PUT request adds new content to a specific resource and also supports no additional parameters. To perform more complex DELETE and PUT operations can be used the XQuery Update Operation [57].

[15] http://docs.basex.org/wiki/JAX-RX_API

3.3.2 Implemented Services

To make use of the BaseX JAX-RX API were created five Java classes to implement the REST operations, which can be observed in figure 3.11.

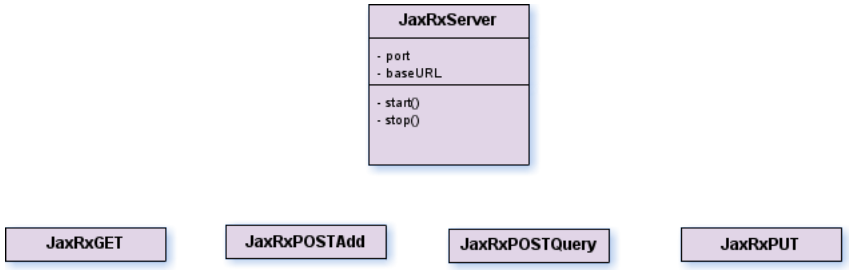

Figure 3.11: Repository Manipulation - Rest API Classes

The JaxRxGET class allows to use all query parameters directly within the URL, consisting the Java class in creating a HTTP connection and construct the URL with the parameters passed by the user. Figure 3.12 shows and example of a possible URL with the option query (which allows to define a XPath/XQuery expression to query the database) with a query to the EHR10000R database to get the first three electronic health record ids. The POST

Figure 3.12: Repository Manipulation - Get Query Example

method is divided into two options. These options differ in the value given to the content-type attribute, which can take the value text/xml or query+xml. The JaxRxPOSTAdd class is used to add new documents to a database or collection. To accomplish this the content-type attribute is set to text/xml. The HTTP request body will be added as new XML document to the specified database. To execute queries (JaxRxPOSTQuery) the content-type hast to be set to query+xml. By doing this the HTTP request body is interpreted as query. The body must be constructed according to the JAX-RX POST Schema [16]. In both cases the HTTP response is 200 (OK) in case of success. Finally the JaxRxPUT class is used to create or update a database resource. The HTTP response is 201 (created) or 404 (not found) if there is an error. The JaxRxServer class contains the basic configurations (port and baseURL of the server location) and the functions that allow to start and stop the server. The root URL (Figure 3.13) lists all available databases. The base URL allows to perform several actions by

Figure 3.13: Repository Manipulation - Base URL

adding specific parameters. For instance, the documents of a database can be listed adding the database name to the base URL or contents of a database can be accessed adding the query command to the URL. POST and GET methods can be extended with different param-

[16] http://docs.basex.org/wiki/JAX-RX_POST_Schema

eters like **query** (evaluates and XPath/XQuery expression), **run** (runs a query file located on the server) or **command** (executes a database command).

All the implemented services were implemented using SOAP, being defined in WSDL files, allowing the re-use by any other application. The functionalities can be divided into four groups according to their area: DBManager, ServerManager, EHRManager and QueryService, as depicted in figure 3.14.

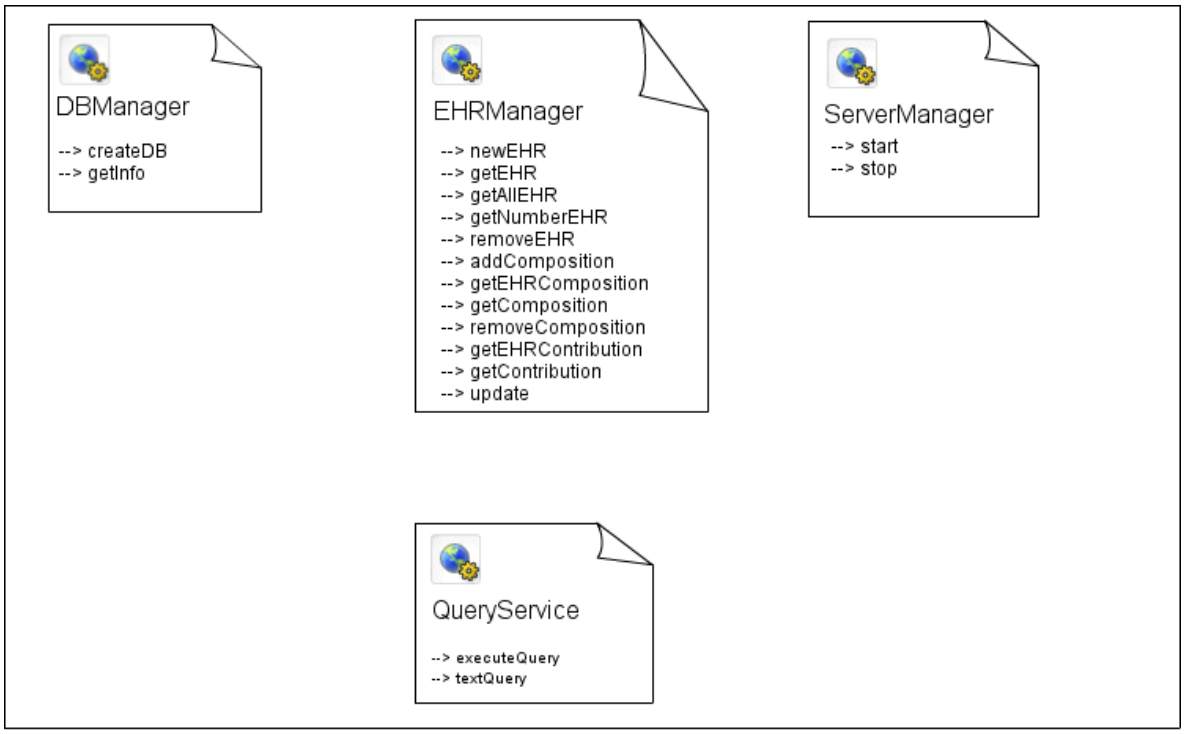

Figure 3.14: Repository Manipulation - Web Services

The DBManager file contains the services responsible for the administrative operations on the repository, such as creating a new database or obtaining information about a already existing database. The two services offered to fulfil the administrative requisites are CreateDB and GetInfo. **CreateDB** allows the user to create a new database. The parameters to pass are the database name and the path of the XML file to initiate the database (this field can be empty to create an empty database). To create the database is used the JaxRxPUT class, which adds a XML String to a specified path. By indicating a path (the name of the database to create) and an empty String a new database is created. The **GetInfo** service devolves database information, such as which databases are created, with how many nodes, creation date, etc. This is done using the JaxRxGET class with the command parameter. This parameter allows to execute commands on the server, being in this case the LIST command executed. LIST is a BaseX command which lists all available databases, or the documents in a database.

The EHRManager file is responsible for containing all the Web Service available for record manipulation. This set of services cover the requisites regarding the information management

of electronic health records, being them the insertion/edition/deletion of EHRs, compositions and contributions. The **newEHR** service allows to insert a new Electronic Health Record into the repository (it receives a XML String with the medical information to insert). First is extracted the EHrID from the XML and made a query to the repository to check if it is not already inserted. If this is not the case the data is inserted using the JaxRxPOSTAdd class (HTTP body is interpreted as information to add by the server). The **getEHR** service returns the EHR which corresponds to the ID passed as parameter. It uses the JaxRXGET class with the parameter *query*. The **getAllEHR** service returns all records existing in the repository. Similar to the getEHR service it uses the JaxRXGET class with the parameter *query*. To retrieve a limited number of EHR's there is the **getNumberEHR** service, which devolves the first N (being N specified by the user) records of the repository. The **removeEHR** services allows to remove an EHR, specifying an ID. This is done using the JaxRxPOSTQuery class where the HTTP body is interpreted by the server as a query. The corresponding query is shown in figure 3.15, where the *id* is passed to the service by the user. The **addComposition** service adds a composition to an existing record. To do this

```
"<![CDATA[delete node \n"
 + "myEHR[ehrID/value='id']]]>"
```

Figure 3.15: Repository Manipulation - Remove Query

the user has to indicate the id of the record to which the information will be added. First the record is validated according to the OpenEHR Composition schema. To do this validation is used the javax.xml.validation API, as explained before. If the information follows the indicated structure it is verified if the record has already a Compositions tag to which the new composition can be added. If not, it is created using the JaxRxPOSTQuery class. Then it has to be created a contribution which reflects the change that will be made to the record. Finally the composition and created contribution are added to the record using the JaxRxPOSTQuery class. The **getEHRComposition** service returns all compositions of a record specified by the user via ehrID. It uses the JaxRxGET class. The **getComposition** returns a composition identified by the EHR id and the composition id, using the JaxRxGET class. The **removeComposition** service removes a composition identified by the EHR id and the composition id. It uses the JaxRxPOSTQuery with a query similar to the one used by the removeEHR service. The services **getEHRContribution** and **getContribution** are similar to getEHRComposition and getComposition respectively. The **update** service allows to update any node of the XML structure, using the W3C update [58]. The user specifies the path of the node to change and a new value to replace the existing one. The update is done using the JaxRsPOSTQuery class. The fact that BaseX supports this kind of information update brings major performance improvements, as it allows to access a specific node and to edit its value. Otherwise it would be necessary to read the whole file, edit the information and rewrite the file, which would cost much more time and resources.

The QueryService allows to query the data available in the repository. The two available services to retrieve information are executeQuery and textQuery. There are two standardized query languages for XML, XQuery and XPath, which are both powerful for querying and navigating the structure of XML. The **executeQuery** service simply receives a query written by the user according to these standards and resends it to the server using the JaxRxGET

class with the option query. BaseX interprets the query and devolves the results in XML format. However, none of the query standards mentioned supports fully the full-text search. To solve this emerged a new standard: XQuery and XPath Full Text 1.0 (XQFT) to address Information-Retrieval issues. To add support for full-text search, XQFT extends XQuery and XPath in three ways. It adds new expressions as *ftcontains*, which evaluates a sequence of items against a full-text selection, it enhances the syntax of FLWOR expressions with score variables (to evaluate the relevance of the result to the search condition) and adds full-text match options to the static context, which can be declared using *declare ft-option* $<$*match option* $>$ before the query. BaseX is an early adopter of the XQuery Full Text Recommendation, extending it with some useful functions, as ft-search (performs a full-text index request and returns all text nodes which contain the specified text) or ft:count (returns the number of occurrences of search terms specified in a full-text expression). BaseX supports two kind of index structures which allow a very efficient search: Compressed Trie (for standard full-text queries) and a special fuzzy index for fuzzy queries. To make use of the full-text query option is used the **textQuery** service which allows free text queries. This means that a user indicates the text to search for and it is searched for that text in all values of the XML file. This is done by using the BaseX's ft-option like showed in figure 3.16.

```
"declare ft-option using case sensitive using stemming;"
 + "myEHR[//*[text() contains text '" + text + "']]";
```

Figure 3.16: Repository Manipulation - Full-Text Query

The evaluation method of the query is chosen automatically by the BaseX processor according to the nature of the input data and the created indexes. There are three available methods: sequential scan (performs the predicate test for each location path), index-based processing (performs the predicate test first and traverses the inverted path for all index items) and a hybrid approach (combination of sequential and index-based processing) [50]. To improve performance of the full-text query option, was created a full-text index, with the options case sensitive and stemming ON. The XQFT also allows the usage of scoring models and values within queries, with scoring being completely defined by the specific implementation. BaseX has an internal scoring model which can be easily extended to different applications, allowing additionally to store scoring values within the full-text structure. There are three scoring types available [17]:

- 0: standard algorithm is applied (considers the length of a term and its frequency in a single text node)
- 1: standard FT/IDF algorithm (treats documents nodes as documents units)
- 2: TF/IDF algorithm where each text node is treated as a document unit

To create the full-text index structure was used scoring option 2, as these variant is recommendable for large XML files which contain only one document node. For the implemented repository it consists in the best choice, as there is a single root node which contains an EHR node for each patients electronic health record.

[17] http://docs.basex.org/wiki/Full-Text

3.4 Application Layer

To demonstrate the functionalities of the business layer there was developed a web application which shows a possible usage of the implemented services. This web interface was designed to fulfil the requisites of an EHR management system, allowing to:

- consult database information
- insert/edit/remove EHRs
- view EHR content
- get Composition list associated to an EHR
- insert/edit/remove compositions
- visualize formatted XML content
- get Contribution list associated to an EHR
- query database (XPath and free text)

To develop an interface that supports these functionalities were considered a few development frameworks which will be discussed in the next section.

3.4.1 Web Development Frameworks

A Web Development Framework is a software framework that is designed to support the development of websites. Many frameworks provide libraries for additional functionalities. They differ in many aspects such as client or server-side business logic, the way data is exchanged between client and server-side or time of responsiveness. jQuery, ZK, Google Web Toolkit (GWT) and Stripes are four open-source Java-based frameworks which will be discussed.

jQuery

JavaScript is a programming language based on ECMAScript standardized in the ECMA-262 and ISO/IEC 16262 specifications [18]. It is very often used to program client-side operations in web browsers. It was conceived to be an object oriented script language based on prototypes. jQuery [59] is a JavaScript library that simplifies HTML document traversing, event handling, animating and AJAX (Asynchronous JavaScript and XML) interactions for web development. This library is light-weighted and small compared to other JavaScript frameworks. It has a very large community so there is is wide range of plugins available for specific needs and very good documentation. It is very easy to extend and learn. One of the great advantages of jQuery are the very powerful chaining capabilities. It makes it possible, for instance, to start with a table, drill down to find cells with a specific class and change their background attribute. The whole concept of querying and chaining fits very well with DOM manipulation, which seems to be what JavaScript libraries are mostly used for. Two other advantages are cross-browser compatibility and the separation of JavaScript code from

[18] http://www.ecma-international.org

HTML mark-up. The disadvantages of jQuery can be the need of a server framework for initialization and session management and the need of another application to supply data (like a Web-Service). It is also not easy to debug and the source code is difficult to protect. In some cases functionality may be limited. Depending on how much customization is required, the use of raw JavaScript may be required. In sum, the advantages heavily outweigh the negative effects of using jQuery so it is without a doubt a winner for developing an interface using JavaScript commands.

ZK

ZK [60] is an Ajax + Mobile framework. It is an open-source, Java-based web development framework, released under the LGPL license. It includes an Ajax-based and event-driven engine (allowing intuitive programming), rich sets of XML User Interface Language (XUL) and HTML components and a mark-up language (ZK User Interface Markup Language (ZUML)) which makes the design of rich user interfaces as simple as authoring HTML pages. With these features it enables web applications to have rich user experiences. Furthermore it allows scripts in Java, supports multiple browsers and look and feel are controlled by Cascading Style Sheets (CSS) (so it is very customizable). ZK is almost completely server-side. This has advantages and disadvantages. On one hand it is possible to access all background resources straightforward, one the other it results in a slower responsiveness. To compensate ZK allows to write some code at the client to enhance responsiveness of the critical parts (this has to be done using JavaScript). The framework dynamically skins code as JavaScript data sent to the browser over Ajax. ZK needs no compilation. It detects and reloads modified pages. It has a lot ready to use widgets and also allows to create custom components. Although a drawback is that some of the components (for example the fish eye bar) are only available in the commercial version. It has also a good documentation and a strong marketing machine which is good for users, since the commercial aspect is served by building a strong community.

GWT

GWT [61] is a development toolkit for building and optimizing browser-based applications. GWT is client-side only technology, which means that business logic is exposed at client side. This has the advantage of a much faster responsiveness (less client-server requests). On the other hand this means that there has to be marshalling between client and server (and GWT RPC/JavaScript Object Notation (JSON) supports only very simple objects). GWT is different from other frameworks as the developer writes both client-side and server-side code in Java. The compiler transforms the client-side code into browser-compatible JavaScript. This has clearly the advantage of no JavaScript errors. Java is a strongly-typed language which is easily debugged. Also with Java there is the possibility to use complex data types (such as HashMaps or ArrayLists) on the client-side. Data can be send using these types to and from the server. But the lack of JavaScript can also be considered a disadvantage. There is no way to put JavaScript in the HTML code. For that is used JavaScript Native Interface (JSNI), to wrap JavaScript in Java. This is very powerful but difficult to do.

Stripes

Stripes [62] is a presentation framework for building web applications using Java technology. It is an open-source, action-based Java web application framework related to the Model

View Controller (MVC) pattern for web development. Through Java technology it manages to be very light-weighted. For instances is uses annotations in spite of configuration files (convention over configuration). To get started there is no external configuration needed. Its main advantage is simplicity. With its clean design and extensibility it is very transparent and easy to use.

Web Development Framework Assessment

There are numerous web application frameworks and just four have been discussed. All of them have their advantages and disadvantages. It is for the user to analyse his applications needs to choose which one (or ones) fit best. From this four, jQuery seems to have the most advantages and functionalities for programming web applications. Combined with a framework more related to the look of the application (such as ZK) there can be developed very functional and beautiful web applications. Regarding the needs of the interface to develop was chosen ZK with help from JavaScript and jQuery for some specific operations.

ZK is an open-source web development framework, written in Java, which includes an Ajax-based event-driven engine, rich sets of GWT and XHTML components and a markup language. Although it is easy to create good looking web pages and implement simple operations using ZK, there was still the need to fall back to Javascript for some specific functionalities. ZK follows a server centric approach, being the synchronization and event pipelining processed automatically by the ZK engine. This makes the AJAX calls completely transparent to the web application developer. To create web pages ZK supports the Markup language ZUML that enables the definition of rich user interfaces. ZUML is based on XML and has the extension zul. It is also possible to include Java code in this files, using the zscript tag.

3.4.2 Development

To develop the web application were used several technologies, to implement all functionalities projected, being ZK the web development framework used. Figure 3.17 shows the architecture of ZK. The Ajax-based mechanism of ZK involves three important parts: ZK update engine (asynchronous), ZK loader and ZK client engine. The ZK loader and ZK update engine are composed by a set of Java servlets and the ZK client engine is composed of JavaScript codes. The communication process is also depicted in figure 3.17. When an URL request is send by the client-side (browser) to the server, it is interpreted by the ZK loader (server-side), which then creates a corresponding HTML page (including standard HTML, CSS, JavaScript and ZK components). The created page is send by the ZK loader to the client and the ZK client engine, which is located at the client side and is used for monitoring JavaScript events queued in the browser. As shown, when an event occurs, originated by interaction of the user with the created page and its DOM elements, it is send a notification to the ZK widgets and then to the ZK client engine. The ZK client engine only re-sends those AJAX requests back to the ZK update engine on the server side. The ZK update engine interprets the request, updates the ZK properties and sends a response back to the client. With the response received, the client can than update the corresponding content in the browser's DOM tree. The process is repeated until the URL is no longer reference by the user [60]. This mechanism has the advantage that the synchronization of the states of the components between the browser and the server is done automatically by ZK, and transparent to the application.

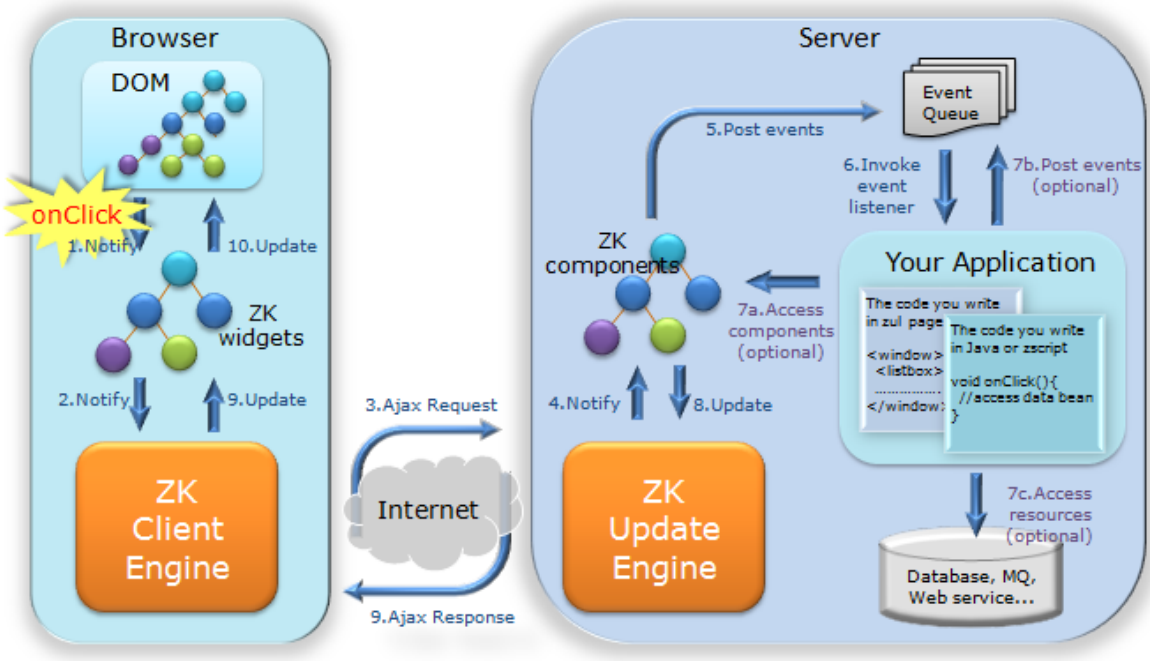

Figure 3.17: Web Application - ZK Architecture from [63]

As already mentioned, ZK runs a set of Java servlets inside a Java servlet container. To run ZK and Java servlet container it was necessary to have the latest version of JRE (Java Runtime Environment) installed. In this case was used the JDK (Java Development Kit), which includes JRE, a compiler and a debugger, in addition with the Netbeans IDE. For the Java servlet container was used Apache Tomcat. The developed application consists in several files: ZUML files, Java classes and JavaScript and jQuery functions.

ZUML files

The markup language ZUML allows the user to create ZK components by simply declaring an enclosing tag (with format similar to HTML tags). Pages build by ZUML should have the extension *.zul so they can be interpreted by the web server. The load and interpret phase of a ZUML page consists in four phases [60]:

- page initialization
- component creation
- event processing
- rendering

The **page initialization** phase consists in running init processing instructions. This makes it possible to specify a class of which a instance is created and its doInit() method is called as an initiator of the page. If no processing instruction is defined, this phase is skipped. In the **component creation** phase is interpreted the ZUML page by the ZK loader, which means the creation and initialization of the components required by the page. This implies

several steps, consisting the first in the examination of the *if* and *unless* attributes. It they are false the element and all of its child elements are ignored. Next is created the component, according to specified element name or class, and the members of the component's class are initialized. All this is done by the ZK loader, which also interprets any nested elements and repeats the whole procedure. Finally, is invoked the *afterCompose* method (for this to happen the component must implement the *AfterCompose* interface. When all children are created, is sent the *onCreate* event to the component, making it possible that new elements can be initialized later by the application. In the **event processing** phase are invoked the listeners for each event queued, being each listener invoked by an independent thread. In the last phase (**rendering** phase) ZK renders the component into a regular HTML page and sends it to the browser. This is done by calling the *redraw* method.

JavaScript

To permit the XML visualization of the data representing an EHR was used an already existing JavaScript set of functions and adapted according to the specific needs of the application. The used JavaScript file offers the possibility to create an user friendly visualization of XML data within the ZUML page. The XML display is dynamic, so the user can expand and collapse any node. The function called (*LoadXML*) receives originally as parameter the ID of the HTML element intended to contain the displayed XML. To meet the applications requisites were added three parameters: a String containing the XML to display, a String containing the word to display in a different color (to display query results) and a flag to indicate if the text values are to be shown as plain text or within a text box (for update mode). The two last ones are optional.

Another modification was made to the JavaScript file to modify the send process of update data to the server. To update a value was used the JSON library. JSON is a lightweight data-interchange format and it is used to exchange data. The data is send to a Java Servlet which then calls the right Web-Service to perform the operation. When a user updates more than one field of an EHR the changes are saved in an array and send all together at the end of the operation, so there is just one call to the server for the whole change.

Java classes

For some functionalities were used auxiliary Java classes. Basically they were used to perform two major tasks: manage the data of a list and allow the passing of variables between windows. To manage lists or records were used three Java classes: MyModel (implements *ListModelList*), MyStringComparator (implements *Comparator*) and MyStringRenderer (implements *ListitemRenderer*). This allows to construct an *ArrayList* with the items to show and to create a new list model with it. In the zul file is create a lisbox component, associating the created instances of MyStringRenderer and MyModelList to the attributes *model* and *itemRenderer*. Using the *listhead* tag can be defined the comparators to sort the list (*sortAscending* and *sortDescending* attributes. Additionally is defined the *onSelect* attribute, which contains the code to execute when a item is selected (to know which one can be used the *getSelectedIndex* method.

To pass a value from a window to the next it is stored in a HashMap and retrieved by the other class using the *onCreate* event. This event has an attribute *CreateEvent* which contains the HashMap, that can be retrieved using the *getArg* function.

jQuery

Finally there was also used the jQuery library. jQuery is a JavaScript library that simplifies HTML traversing, event handling, animating and Ajax interactions. It was mostly used to access web components within the JavaScript file. Using the jq('$buttonDiv') command is it possible to access to buttonDiv component created in a zul file from JavaScript and manipulate its attributes. For instance to monitor the text boxes of the XML display (when the user is on the update page) there is the need to access components of the zul file from JavaScript. Here is added an eventListener, called *change* to the text box, allowing to execute a method every time the value within it changes. This method accesses the text box using the jQuery expression *elem = $(#id)*, being id the variable containing the id of the text box. The new value can than be obtained by doing *elem.val()*.

3.5 Results

After explaining the architecture of the developed system, this section will now show the obtained results after the implementation. The implemented web application offers a graphical way to call all Web Services in which consists the developed service layer. In figure 3.18 is shown the initial site of the created web application. The functionalities supported are divided into two areas: Database Management and Query of the Database. The Database Management includes administrative operations as well as all operations necessary for the management of the records (insertion, update and removal of OpenEHR electronic health records and their compositions and contributions). The Query section allows to retrieve the inserted information, being it possible to query whole records, attributes or doing text searches. On the left side (1) the user can switch between database management and Searching

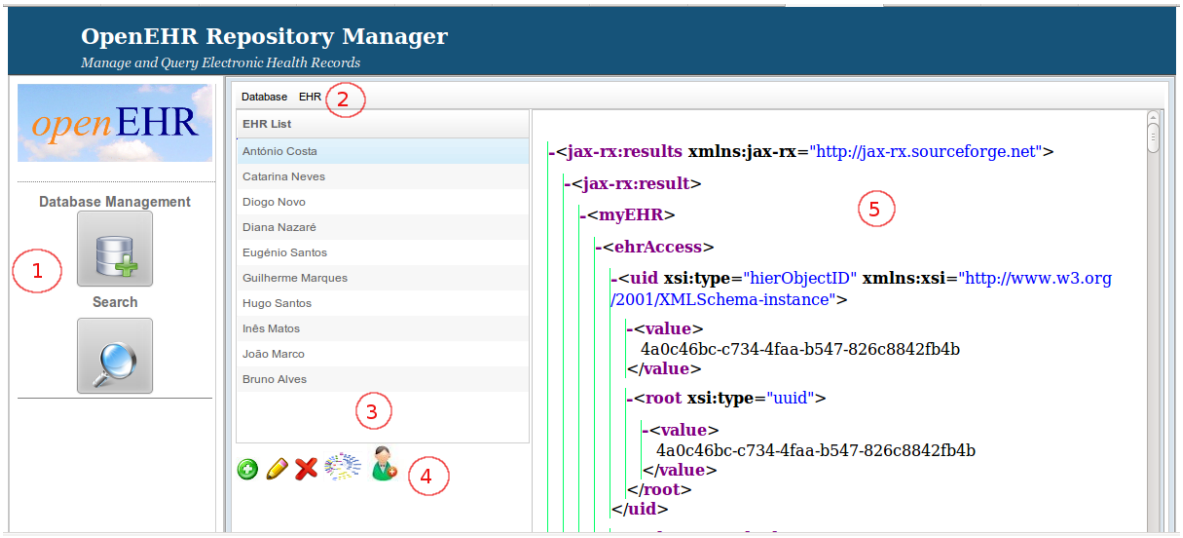

Figure 3.18: Implementation Results - Main interface

the database, using the depicted menu. In the figure is selected Database Management option. Within this option it is possible to select between two tabs (2), being showed in the figure the tab EHR. This allows the user to visualize the records inserted into the database, which appear in area (3). When a record is selected (in this case the record of patient António

Costa) his information is displayed in area (5) in the format of XML data. The XML is displayed dynamically, being it possible to collapse or expand any node. To manage the records the users has the buttons in area (4) which allow to add, edit and delete records, as well as visualize compositions and contributions of the record. Each of this operations will be explained in more detail in the following two sections.

3.5.1 Database Management

The Electronic Health Records are added uploading the XML files which contain them. This files have to follow the structure of the OpenEHR standard to be accepted. The interface allows to upload these files, pressing the add button, as shown in 3.19. This process calls the addEHR web service, passing the String read from the uploaded file. The new added

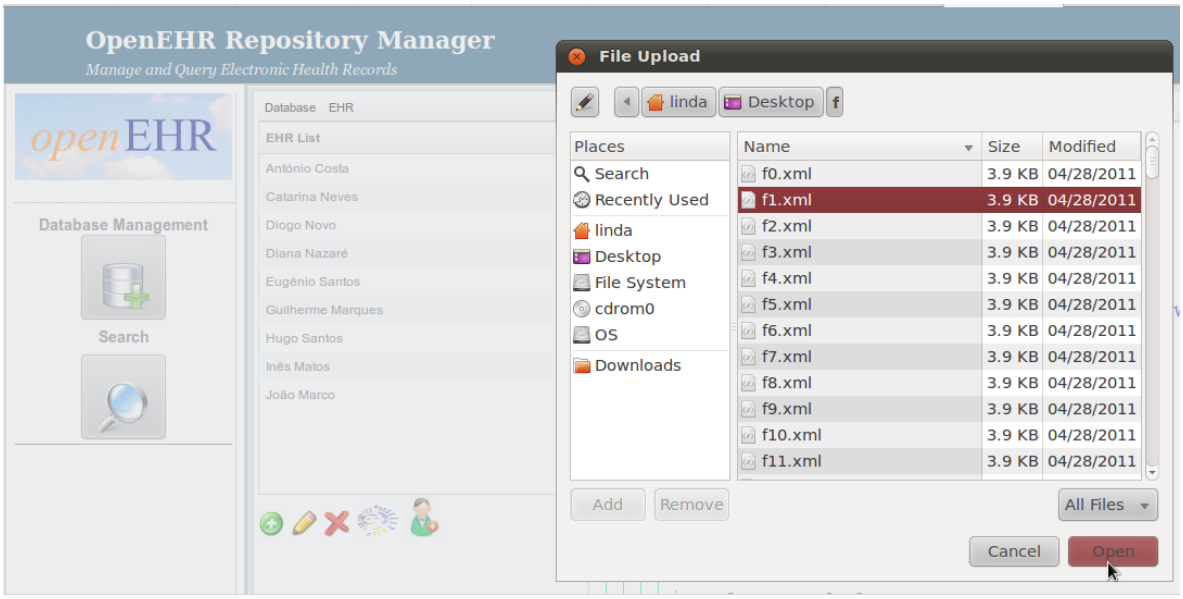

Figure 3.19: Implementation Results - EHR Upload Window

EHR is inserted into the list of existing records in the database. When the edit button is pressed the XML view changes, transforming the representation of all text values from simple text to text within a text box. The user can edit as many as he wishes. To finalize the edition he has to press the Submit button. This way there is only send one request to the server, independently of how many fields have been changed. It is also possible to delete an EHR or one of the compositions associated to it. To do this is read the ID of the selected item and made a request to the removeEHR Web Service, passing the ID as parameter. The list model is immediately updated to show the list without the just deleted record. Each record has associated a list of compositions. This compositions represent the actual medical information. The Composition button allows to view the list of compositions of each patient. The insertion mechanism is similar to the EHR insertion process, being done by uploading the XML file. All compositions are validated according to the OpenEHR definition, via XSD files. If a composition is not valid an error message occurs (figure 3.20). In this case the composition is missing the language tag, being the insertion process aborted. The error message gives indication of which attributes are missing, allowing the user to easily change

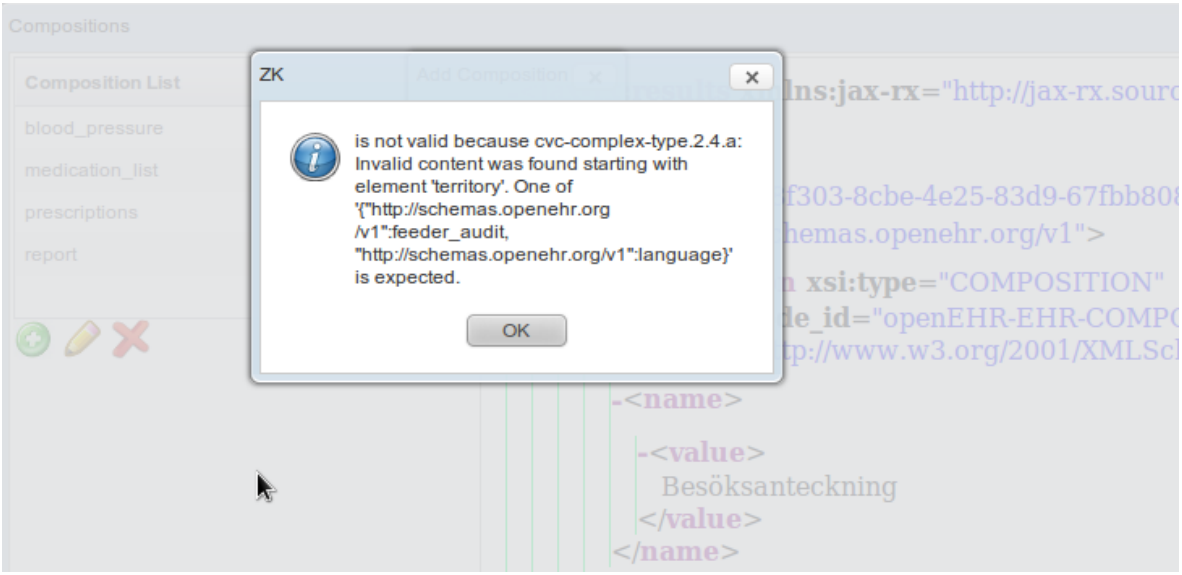

Figure 3.20: Implementation Results - Composition Validation Error

the composition to be a valid one. When clicking the *Database* tab, the user has access to database specific information. Here he can also change the Base URL, which allows to specify which repository to use. Other database management operations are also possible, as resetting the chosen database or starting/stopping the BaseX server.

3.5.2 Query Database

There are two available query methods: free query and full-text query. The Free Query option allows the user to run any query in the XQuery/XPath format, covering so all types of queries, from selecting one patient's record to counting how many patients did have a radiology exam on a specific day. The text query allows to search for a string within the value elements of the XML. To limit the search the user can specify the XPath to the elements in which he wishes to search (Figure 3.21) In this example it is searched for the openEHR token in all myEHR/ehrAccess nodes. The text values which contains the indicated substring are displayed in green to mark them as results of the executed query. The path choice is made easier by an auto-completion function (Figure 3.22). This is done by dynamically creating an *ArrayList* with all possible paths to create a instance of the *SimpleListModel* class with it. This model is than added to a *combobox* component, setting its *autodrop* attribute to true. This leads to an automatic expansion of the items of the combobox, having in sight the already digitized words by the user.

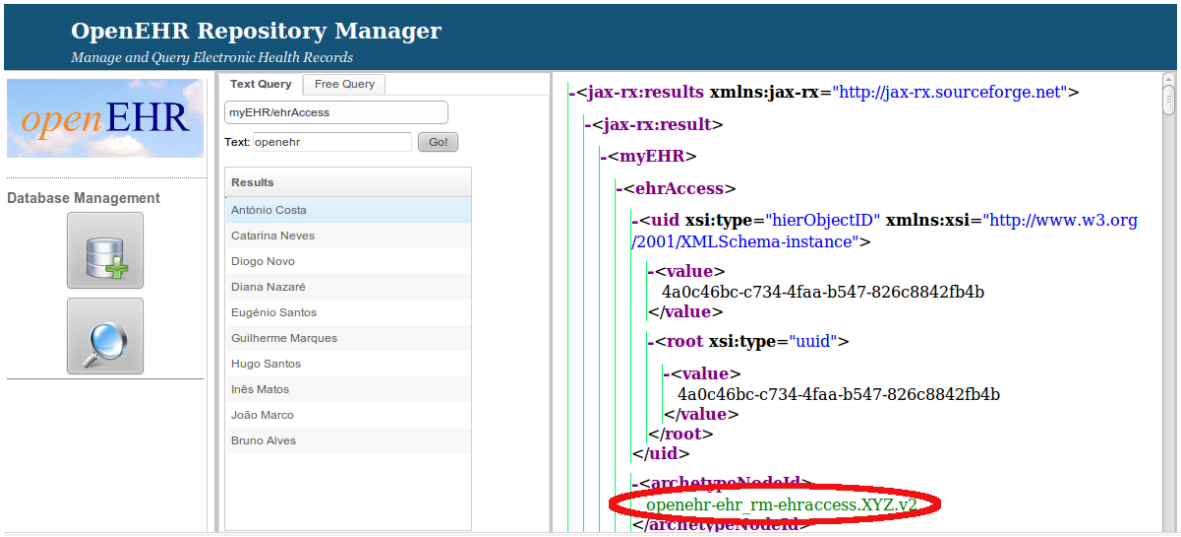

Figure 3.21: Implementation Results - Search results visualization

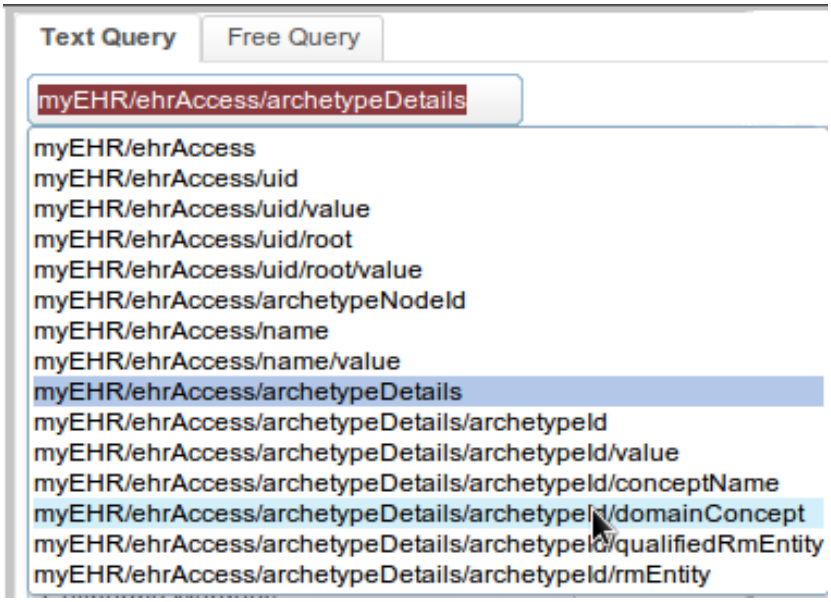

Figure 3.22: Implementation Results - Search Path Auto-Complete

Chapter 4

Evaluation

This chapter describes the performance tests made to the repository and the obtained results. To evaluate the performance there were made test with three sets of data, containing 1000, 10000 and 30000 patient records. The tests were chosen trying to simulate the usual usage of the implemented system.

4.1 Created Indexes

To increase performance BaseX offers a set of indexes which will be discussed next. One of these supported indexes is the Path Summary, used to speed up resolution of location paths (figure 4.1). Besides the depicted index, there is also the possibility to create text, attribute

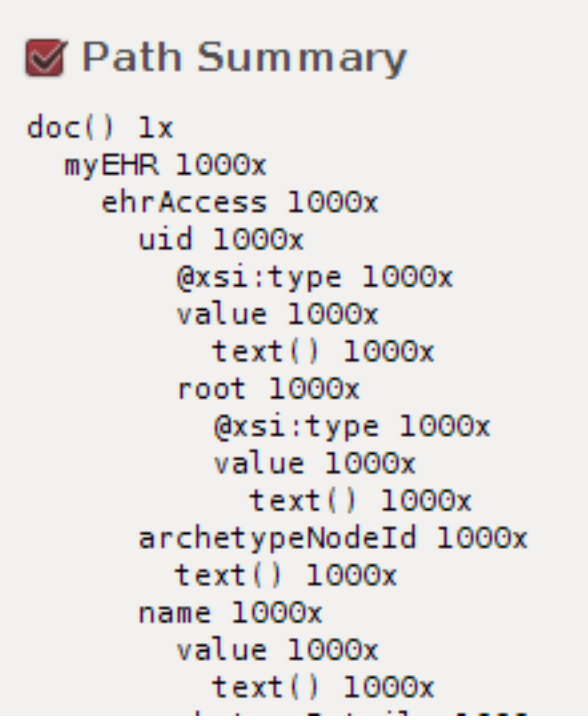

Figure 4.1: Evaluation - Path Summary Index

and full-text indexes. The attributes index (figure 4.2) speeds up comparison between attributes in predicates. As shown, it indexes all node attributes and shows how many times they appear in the document. For instance the internetID attribute appears 1015 times. Tag

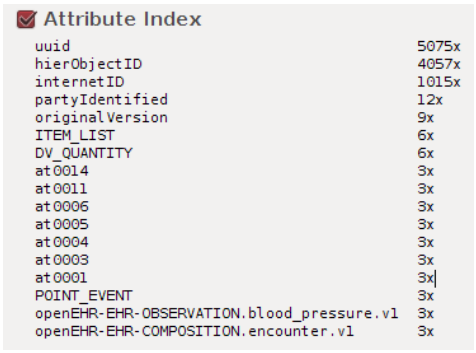

Figure 4.2: Evaluation - Attribute Index

names can also be indexed, as shown in figure 4.3, using a hash structure. A hash index organizes the search keys, with their associated record pointers, into a hash file structure. Text and full-text indexes both index the text values of the XML file, although there are

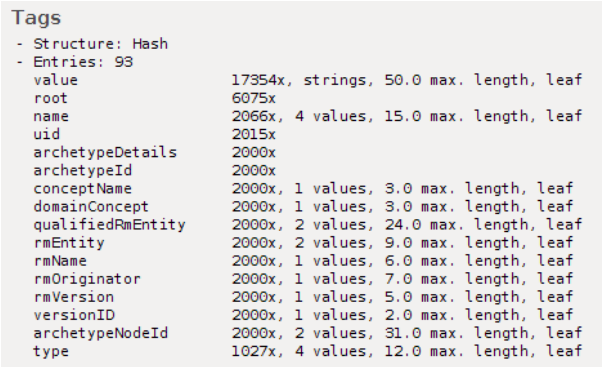

Figure 4.3: Evaluation - Tag Names Index

differences. The text index (figure 4.4) has a binary tree structure and indexes all text values as they are found in the document. The fulltext index (figure 4.5) allows to configure options

Figure 4.4: Evaluation - Text Index

such as stemming, case sensitivity and scoring. The structure is fuzzy which means it is rather

fluid or approximate than fixed and exact.

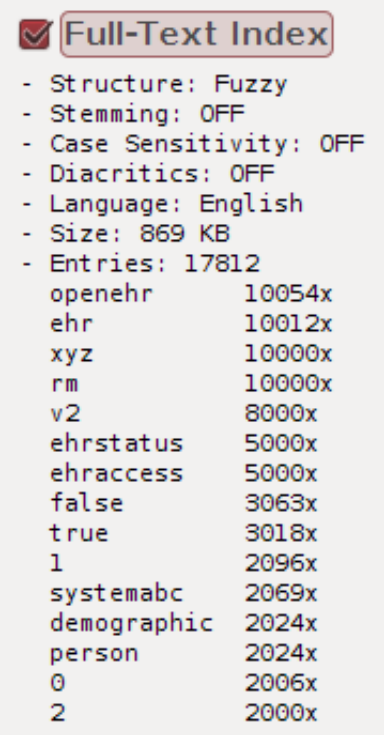

Figure 4.5: Evaluation - Full-Text Index

4.2 Operation Performance

To evaluate the performance there were defined four operations chosen to reflect the posterior usage of the repository:

- Add a new EHR to the database
- Search for a record
- Search for an attribute
- Add a composition to an existing record

All these operations were performed on three databases with 1000, 10000 and 30000 records to evaluate how the performance evolves with an increasing number of records. To build these three databases were created sets of test data. The first approach was to look for already existing instances of the OpenEHR reference model to use "real" health information. Although it was not possible to find this kind of records, as there is no test data made available by the OpenEHR project and real data can not be published for confidentiality reasons. So the solution was to create own test data. To do this were created instances of the OpenEHR reference model, adding random created data. Three sets were created, with 1000, 10000 and 30000 records. The composition test data was offered by a Swedish project, consisting in composition files describing blood pressure and medication list, with Swedish data.

4.2.1 Add new EHR to database

The graph in figure 4.6 represents the time it takes to insert a new record into the repository when it already has a number of records (represented by the x axis). As observable in the figure the time spent inserting a record increases with the number of records already inserted. To insert a record into a database with 1000 records takes approximately 72 mil-

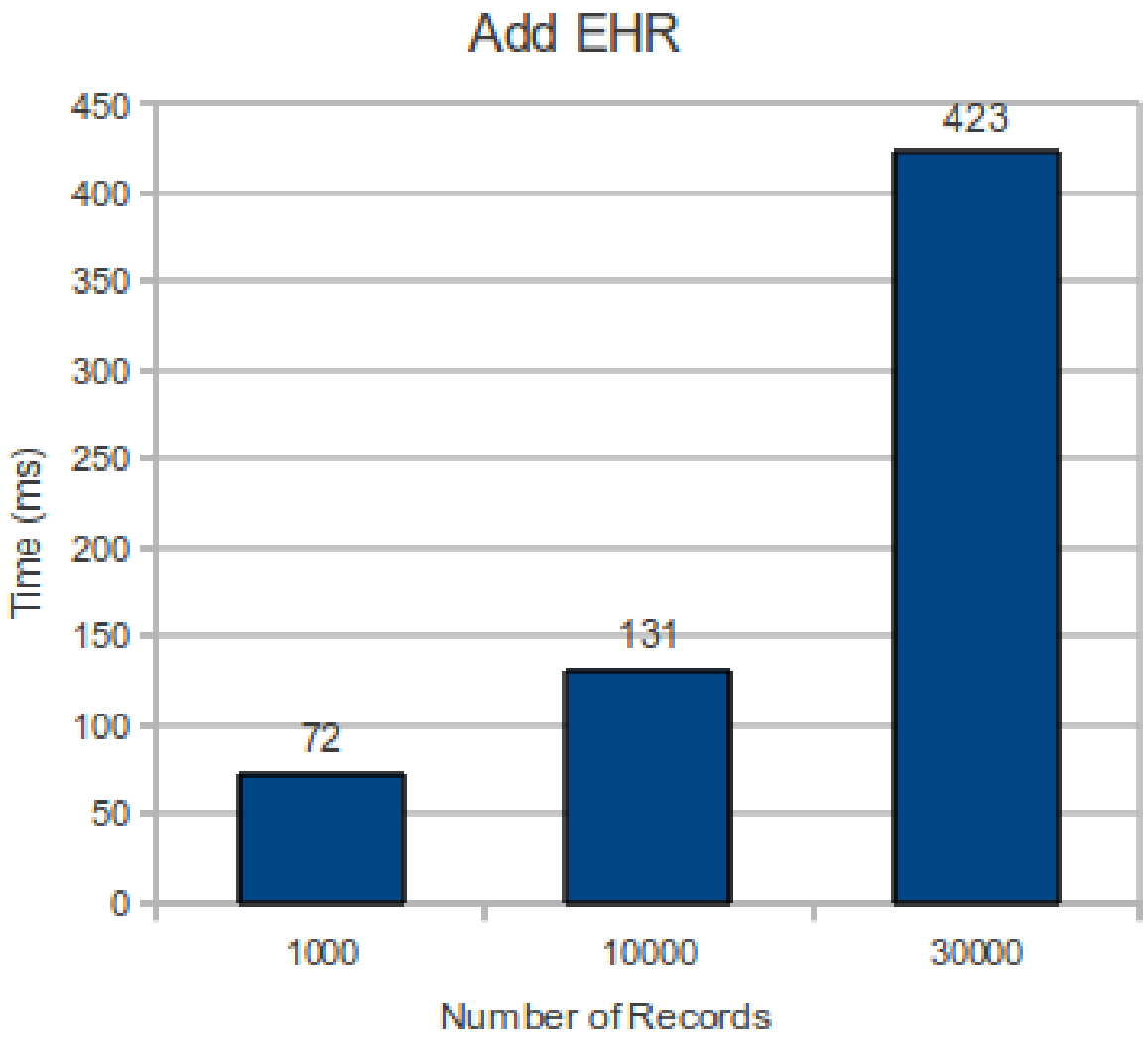

Figure 4.6: Evaluation - EHR Addition Comparison

liseconds. Inserting it into a database containing 10000 (which means with 10 times more records) takes only two times longer. On the other hand between the databases with 10000 and 30000 records the time increases proportionally with the number of records.

4.2.2 Search for a record

This will evaluate how much time will be spent to search for a record given its ID, simulating a clinical operator asking for a patients medical history. Before registering time values of

the database performance it was made a comparison of the times before and after the creation of indexes. To do this was run the query depicted in figure 4.7 against a database containing 10000 records. Figure 4.8 shows the graph representing the resulting times. The first three

```
/myEHR[ehrID/value/text()='94c9d265-61e7-4f3e-9ad2-42df0dfa2e9e']
```

Figure 4.7: Evaluation - Query of EHR comparison

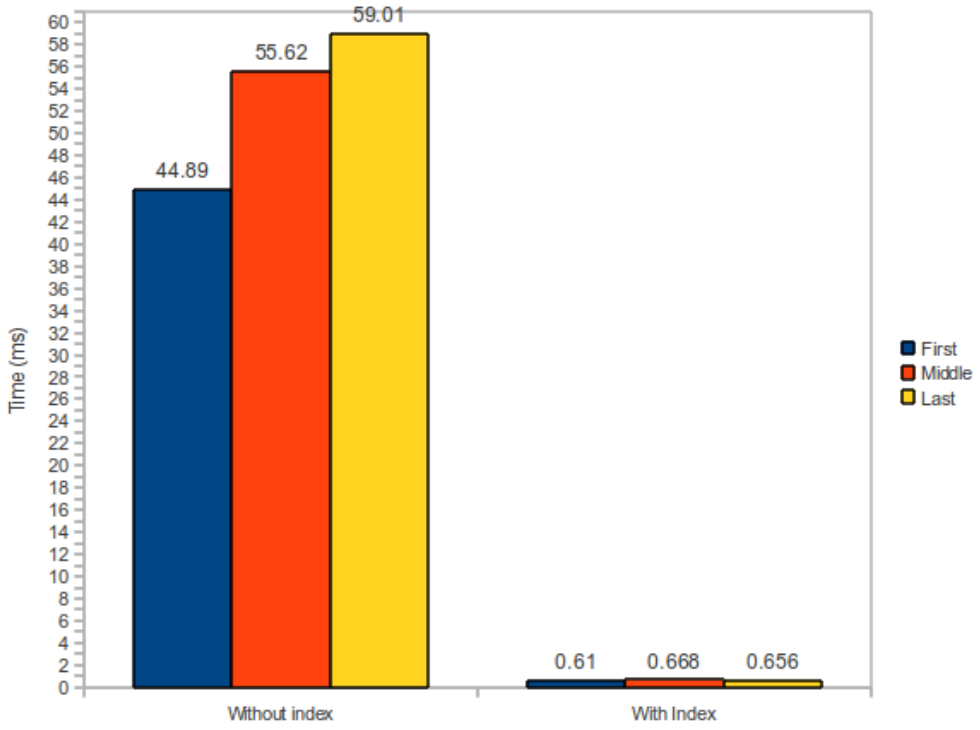

Figure 4.8: Evaluation - Index Performance Increase

bars show the times it took to query the records (first, middle and last) without indexing the database first. The second three bars represent the times registered after the creation of the indexes. Figure 4.9 shows the index used by the BaseX compiler while running the query. The graph shows the acceleration obtained with the use of indexes, having a response

```
- applying text index
Result: document-node { "Ehr1000R" }/db:text("94c9d265-61e7-4f3e-
9ad2-42df0dfa2e9e")/parent::value/parent::ehrID/parent::myEHR[parent::
document-node() { 1003 }]
```

Figure 4.9: Evaluation - BaseX Index Information

time that is approximately 84 times faster. The next test made consists in comparing how the performance evolves with the increase of number of records already in the database. To do this the same query was run against a database of 1000, 10000 and 30000 records. The

result can be observed in Figure 4.10. The conclusion that can be made is that it makes no

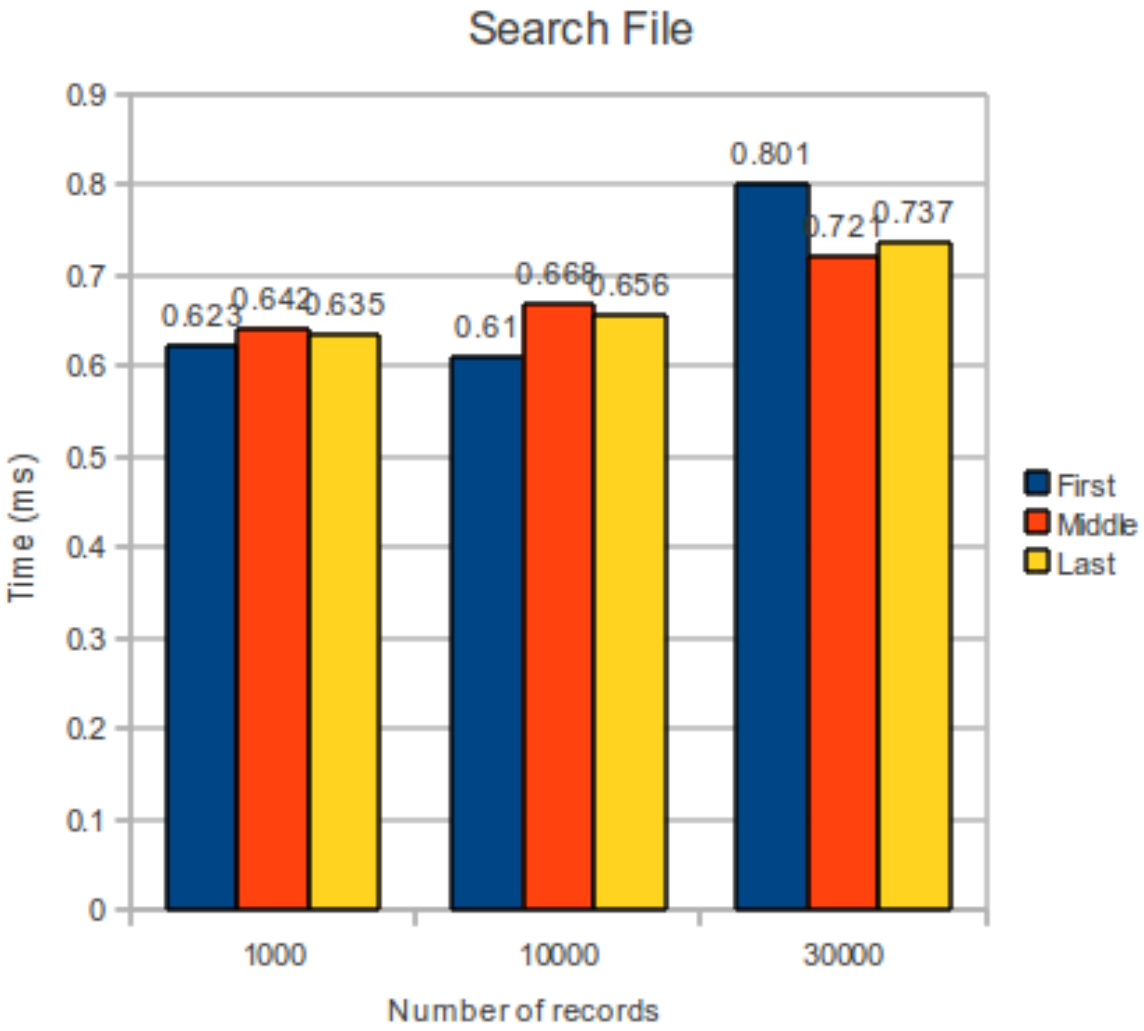

Figure 4.10: Evaluation - Record Search Comparison

difference to search for a record in a database with 1000 or 30000 records, since the response time is constant with approximately 0.66 milliseconds to retrieve a patient's record.

4.2.3 Search for an attribute

To evaluate the performance searching for attributes there were made three tests, increasing by one the level of the attribute in each test. So the search was made for

1. myEHR[ehrStatus/uid/@xsi:type='hierObjectID']
2. myEHR[ehrStatus/uid/root/@xsi:type='uuid']
3. myEHR[ehrStatus/subject/externalRef/id/@xsi:type='hierObjectID']

When searching for an attribute the index used is the attribute index. Again the first test was to evaluate the difference this index makes in terms of response time. To do this the queries were run against a database containing 10000 records before and after creation of the attribute index. The index used by the compiler is depicted in figure 4.11. Figure 4.12 show the graph

```
- optimizing descendant-or-self step(s)
- applying attribute index
Result: document-node { "Ehr30000R" }/db:attribute("hierObjectID")/self:
:xsi:type/parent::id/parent::externalRef/parent::subject/parent::ehrStatus
/parent::myEHR[ancestor::document-node() { 30003 }]
```

Figure 4.11: Evaluation - Attribute Index

that represents the results. This allows to conclude two things. Fist it obviously accelerates

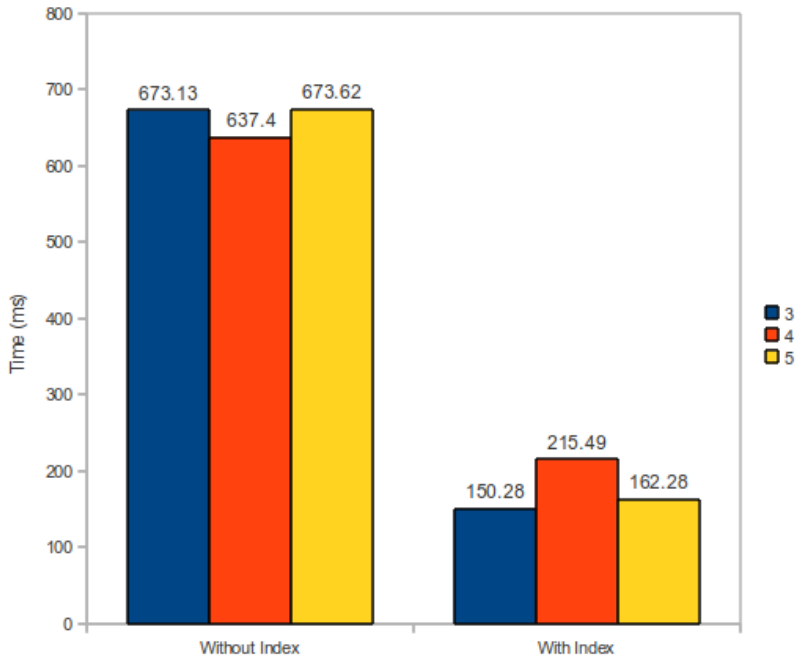

Figure 4.12: Evaluation - Attribute Index Performance Increase

the query time (between 67 and 78 percent). Also it is shown that the index doesn't work the same way for all attributes and it doesn't depend on the level of the attribute. To analyse the evolution of query time with the increase of the database volume the three queries were run against a database with 1000, 10000 and 30000 records. The results are shown in figure 4.13. In opposition to the search of a record, the response time varies with the number of records already inserted, increasing approximately by 0.015 milliseconds per record inserted.

4.2.4 Add a composition to an existing record

To evaluate the performance adding a composition to an existing record were measured the times of the insertion into three databases with 1000, 10000 and 30000 files. Also, in each

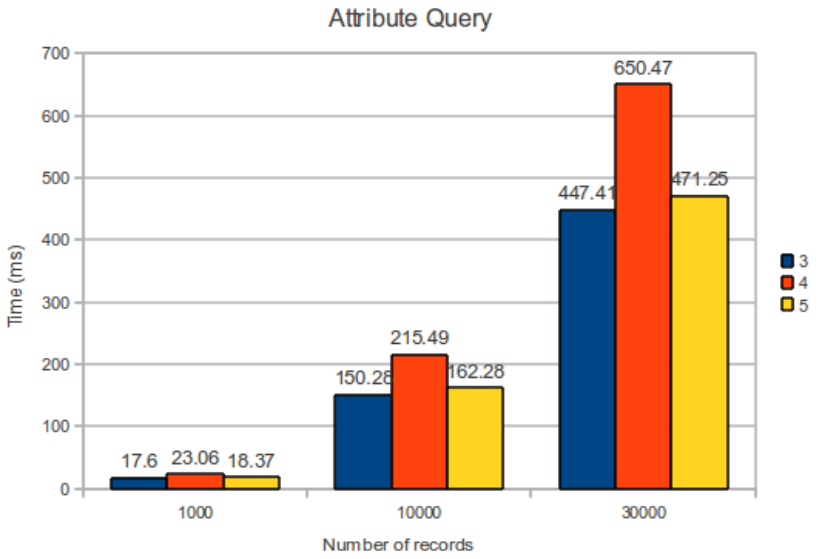

Figure 4.13: Evaluation - Attribute Search Comparison

database, was added the composition three times (top, middle and bottom of the database file). The results can be visualized in figure 4.14. As shown the time of inserting a new com-

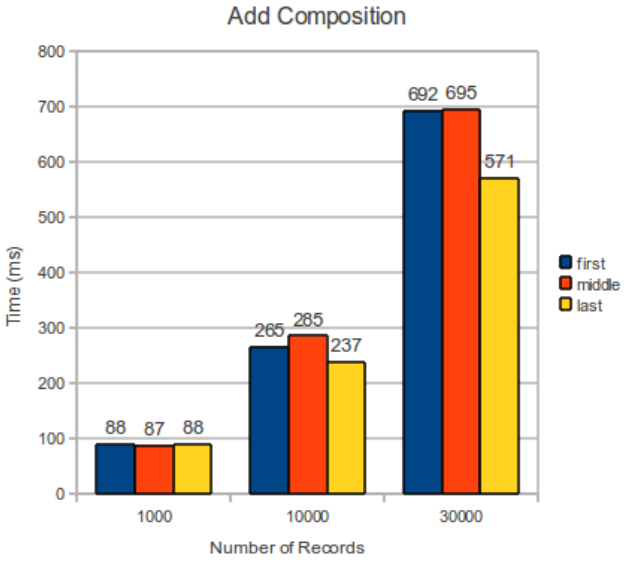

Figure 4.14: Evaluation - Add Composition Comparison

position increases proportionally with the number of records already inserted. The position in which it is inserted does not affect the performance, taking it approximately the same amount of time to add a new composition to an EHR positioned at the top of the XML file as adding it to an EHR positioned last in the XML file in which consists the database.

4.2.5 Database size

Figure 4.15 shows the evolution of the database size with an increasing number of records. The size of the database is insignificant in comparison with the number of records. As shown

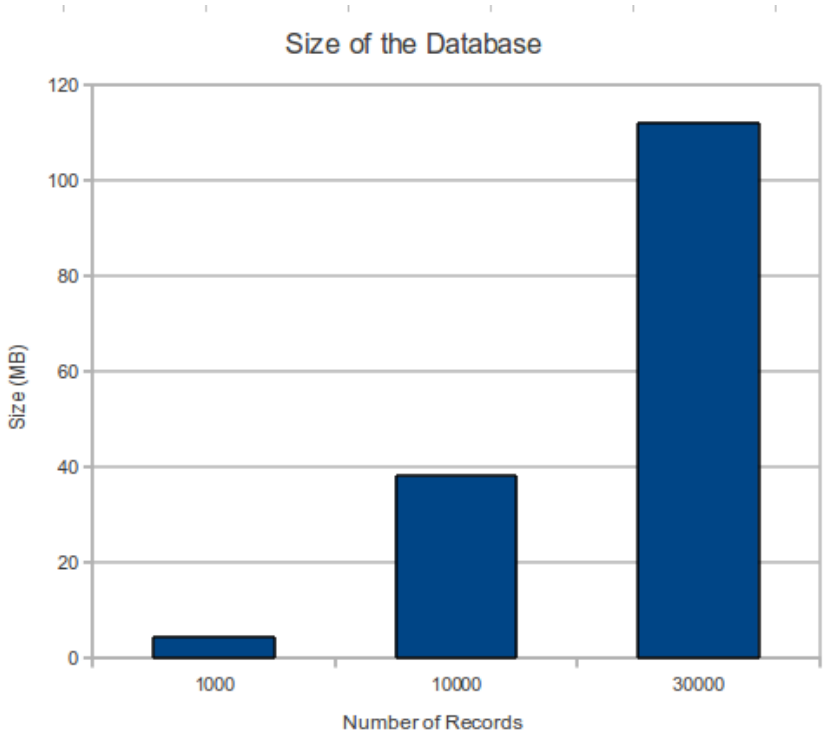

Figure 4.15: Evaluation - Database Size Evolution

30000 records only occupy 112 MB.

4.3 Performance Assessment

In every database system has to be a compromise between insertion and query time. In this case query is notably faster than the insertion process. Analysing the context where the repository will be used (health information systems), it will be faster to query a patient's medical record than to insert a new patient or add new data to the patient's record. This, as everything, has advantages and disadvantages. For instance it can be considered that a patient's record must be created just once, while querying it will happen much more times. On the other hand adding information can also happen repeatedly. What must be considered is that maybe it is not so important for the medical professional to have a fast feedback on that the data has been inserted, but much more to have a quick reply on queries for already inserted medical information.

Analysing the time tables/graphs there can be made some major conclusions. The time it takes to insert a new EHR increases with the number of records already inserted into the database. Index creation influences the response time of queries, making although the text index a much bigger difference than the attribute index. An interesting conclusion is that it makes no difference to query a record from a database with 1000 records and a database with

30000. This makes this repository suitable for systems which require a fast response time for data retrieval. Also it makes no difference what position the record occupies in the XML file. Querying only an attribute is slower when a larger number of records is inserted. This lets conclude that the repository is more efficient for patient centric systems. It is much faster to query a patient's records than to make information centric queries such as, for instance, how many patients have high blood pressure.

Chapter 5

Conclusion

The result of this thesis is an electronic health record repository based on the OpenEHR standard, which uses the BaseX database software to manage the storage of the records and offers a business layer for re-utilization of the implemented services. Additionally was created a web interface to demonstrate a possible usage of the existing functionalities.

Analysing the realization of the thesis, there are some important milestones worth referencing. The first challenge was the full comprehension of the OpenEHR standard. OpenEHR is a very complete and complex standard and not easy to understand for someone new to the area where it is inserted (health information record storage). The specification of the standard is described in several chapters which had to be read very carefully. An important aspect of OpenEHR is its two level modelling approach, being this philosophy what makes the difference between storing OpenEHR records and common information. The two level modelling approach divides knowledge from information. This is done defining archetypes to constrain information, allowing the definition of rules about what data structure and types are valid by domain experts and insertion of information by users who do not have to know all about it. When implementing a repository to store health information this approach brings with it some implications. The main challenge is that it is not possible to define a constant database model, but instead it is necessary to have a flexible repository that accepts every information that was validated against an archetype (existing and future ones). Another fact that added difficulty to the implementation was the lack of available test data. There was no way no validate the designed solution with possible data that would be stored in the repository in the future. All the files that were used to test functionalities and performance had to be generated and filled with fake information.

To design the repository were considered different database technologies. The first approach was the relational model. Due to the always changing data model it would be created just one table, using XML blobs. This way one column contains the XML path to the value and another contains the value itself. This method was dropped not only because of the data dispersion in which it results (it makes it difficult to re-join related data) but also because it is already being implemented by Opereffa, so there would be no new contribution to this research area. Also an option was to store the records in a file system, indexing it using Apache Lucene. This was dropped because of the final solution though it might be interesting to explore this option in the future to do a performance comparison. The technology finally adopted was a native XML database using BaseX. The OpenEHR standard already supports instances of the reference model and archetypes in XML format so it is a natural

choice to maintain this information format and store it in a native XML database. BaseX was chosen because its very efficient indexing capabilities, allowing very fast querying of the information. Also it offers a REST API, fitting perfectly into the SOA approach adopted. Another reason is that native XML database are relatively recent, so it is an interesting case of study to develop an EHR repository using this technology and evaluate the results.

The goal was to obtain a patient centric, flexible health information repository. After its implementation it is possible to say this was achieved, as it supports the insertion of any valid data (according to the OpenEHR specification) into the database, having one EHR per patient containing all its relevant medical information. Although this work consists mainly in a module to use in other applications (repository and service layer) it has the advantage comparing to other solutions that it additionally shows a possible usage of the functionalities. Most of the existing projects focus on a implementation of a part of a health information system based on OpenEHR for integration in other applications. This leads to the fact that it is not possible to see the communication between this parts, to see a possible end result. The web interface does not only show that the functionalities work but can also be an aid for those who wish to reuse the repository, showing them how the services must be invoked.

Future work may consist in adding a security layer for access control and definition of different user roles, being this, for now, left to the application into which the repository might be integrated. Another interesting work may consist in migrate the implemented repository and/or services into the cloud. There are two ways to make use of the cloud technology: cloud computing and cloud storage. Cloud computing has the goal to provision computational resources while cloud storage focuses on online storage of data, distributing it on multiple virtual servers. In the context of the OpenEHR repository this has different consequences. Migrating the BaseX repository and service layer to the cloud (cloud computing) is a relative simple task as there is no need for modification to the current implementation. If the goal is to make use of cloud storage, to have an optimized way to store the EHR, there has to be done a mapping from the current database structure to the format supported by the cloud storage providers. Another aspect that may be considered in the future is the interoperability with other Electronic Health Record standards. Regarding EHR records there can be considered two types of standards: content standards and communication standards, focusing the first ones one how information is structured and stored while the second ones are about information exchange between systems. Being the OpenEHR standards inserted in the group of content standards, it is important to have in mind that there has to be used a second standard if there should be interoperability with other systems. The most used message standard is HL7 which has the particularity that the messages used to exchange information follow a XML structure. As our electronic health records are stored in a native XML database (and consequently in XML format) it might be interesting to explore the possibility of the creation of a HL7 message creator tool.

References

[1] J. Sidorov. It aint necessarily so: the electronic health record and the unlikely prospect of reducing health care costs. *Health Affairs*, 25(4):1079, 2006.

[2] M. Eichelberg, T. Aden, J. Riesmeier, A. Dogac, and G.B. Laleci. A survey and analysis of electronic healthcare record standards. *ACM Computing Surveys (CSUR)*, 37(4):277–315, 2005.

[3] I. Iakovidis. Towards personal health record: current situation, obstacles and trends in implementation of electronic healthcare record in europe1. *International journal of medical informatics*, 52(1-3):105–115, 1998.

[4] MITRE Coorporation. http://www.norc.org/6275/Module9/Electronic Health Records Overview.pdf, January 2011.

[5] T. Beale. Archetypes: Constraint-based domain models for future-proof information systems. In *OOPSLA 2002 workshop on behavioural semantics*, 2002.

[6] D. Kalra, T. Beale, and S. Heard. The openehr foundation. *Studies in Health Technology and Informatics*, 115:153–173, 2005.

[7] L. Bird, A. Goodchild, and Z. Tun. Experiences with a two-level modelling approach to electronic health records. *Journal of Research and Practice in Information Technology*, 35(2):121, 2003.

[8] T. Beale, A. Goodchild, and S. Heard. Design principles for the ehr. *OpenEHR Foundation*, 2002.

[9] T. Beale, S. Heard, D. Kalra, and D. Lloyd. Openehr architecture overview. *The OpenEHR Foundation*, 2006.

[10] T. Beale and S. Heard. Archetype definition language (adl). *OpenEHR specification, the openEHR foundation*, 2005.

[11] T. Beale. Archetype query language (aql). *OpenEHR specification, the openEHR foundation*, 2005.

[12] R. Chen and G. Klein. The openehr java reference implementation project. In *Medinfo 2007: Proceedings of the 12th World Congress on Health (Medical) Informatics; Building Sustainable Health Systems*, page 58. IOS Press, 2007.

[13] M.B. Sanromà, A.V. Mateu, K.G. i Oliveras, and Universitat Rovira i Virgili. Departament d'Enginyeria Informàtica. *Survey of Electronic Health Record Standards*. Universitat Rovira i Virgili. Departament d'Enginyeria Informàtica, 2006.

[14] W.E. Hammond and J.J. Cimino. Standards in medical informatics. *Medical informatics: Computer applications in medical care and biomedicine. New York: Springer-Verlag*, 2000.

[15] A. Hutchison, M. Kaiserswerth, M. Moser, and A. Schade. Electronic data interchange for health care. *Communications Magazine, IEEE*, 34(7):28–34, 1996.

[16] G.W. Beeler. Hl7 version 3–an object-oriented methodology for collaborative standards development1. *International Journal of Medical Informatics*, 48(1-3):151–161, 1998.

[17] M. Poulymenopoulou and G. Vassilacopoulos. An electronic patient record implementation using clinical document architecture. *Medical and care compunetics 1*, 103:50, 2004.

[18] R.H. Dolin, L. Alschuler, S. Boyer, C. Beebe, F.M. Behlen, P.V. Biron, and A. Shabo Shvo. Hl7 clinical document architecture, release 2. *Journal of the American Medical Informatics Association*, 13(1):30, 2006.

[19] P. Marcheschi, A. Mazzarisi, S. Dalmiani, and A. Benassi. Hl7 clinical document architecture to share cardiological images and structured data in next generation infrastructure. In *Computers in Cardiology, 2004*, pages 617–620. IEEE, 2004.

[20] J. Guo, A. Takada, K. Tanaka, J. Sato, M. Suzuki, T. Suzuki, Y. Nakashima, K. Araki, and H. Yoshihara. The development of mml (medical markup language) version 3.0 as a medical document exchange format for hl7 messages. *Journal of Medical Systems*, 28(6):523–533, 2004.

[21] G.V. Koutelakis and D.K. Lymperopoulos. Pacs through web compatible with dicom standard and wado service: advantages and implementation. In *Engineering in Medicine and Biology Society, 2006. EMBS'06. 28th Annual International Conference of the IEEE*, pages 2601–2605. IEEE, 2006.

[22] R. Noumeir and J.F. Pambrun. Images within the electronic health record. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 1761–1764. IEEE, 2009.

[23] Digital Imaging and Communications in Medicine (DICOM). *Digital Imaging and Communications in Medicine (DICOM) Part 18: Web Access to DICOM Persistent Objects (WADO)*. NEMA Publications, 2009.

[24] PG Drew and C. Carr. Integrating the healthcare enterprise. *MD COMPUTING*, 17(3):57–57, 2000.

[25] R. Noumeir. Integrating the healthcare enterprise process. *International Journal of Healthcare Technology and Management*, 9(2):167–180, 2008.

[26] T. Aden and M. Eichelberg. Cross-enterprise search and access to clinical information based on ihe retrieve information for display. In *International Congress Series*, volume 1281, pages 986–991. Elsevier, 2005.

[27] A. Dogac, V. Bicer, and A. Okcan. Collaborative business process support in ihe xds through ebxml business processes. In *Proceedings of the 22nd International Conference on Data Engineering*, page 91. IEEE Computer Society, 2006.

[28] A. Dogac, T. Namli, A. Okcan, G. Laleci, Y. Kabak, and M. Eichelberg. Key issues of technical interoperability solutions in ehealth. In *Proceedings of eHealth 2006 High Level Conference Exhibition*, 2006.

[29] UN ECE and European Committee for Standardization. *Activities of the European Committee for Standardisation (CEN): Note*. UN, 1994.

[30] M. Eichelberg, T. Aden, J. Riesmeier, A. Dogac, and GB Laleci. Electronic health record standards-a brief overview. In *Information & Communications Technology, 2006. ICICT'06. ITI 4th International Conference on*, pages 1–1. IEEE.

[31] D. Kalra. Electronic health record standards. *IMIA yearbook of medical informatics*, 2006:136–144, 2006.

[32] W.D. Bidgood and S.C. Horii. Introduction to the acr-nema dicom standard. *Radiographics*, 12(2):345, 1992.

[33] D.A. Clunie. *DICOM structured reporting*. PixelMed Publishing, 2000.

[34] E.A. Marks and M. Bell. Service-oriented architecture (soa): A planning and implementation guide for business and technology. 2006.

[35] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H.F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1, 2000.

[36] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, 2001.

[37] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, Inc. New York, NY, USA, 1999.

[38] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. Object-oriented modeling and design. 1991.

[39] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, volume 57, 1989.

[40] AM Alashqur, S.Y.W. Su, and H. Lam. Oql: a query language for manipulating object-oriented databases. In *Proceedings of the 15th international conference on Very large data bases*, pages 433–442. Morgan Kaufmann Publishers Inc., 1989.

[41] S. Gao, C.M. Sperberg-McQueen, H.S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3c xml schema definition language (xsd) 1.1 part 1: Structures. *World Wide Web Consortium, Working Draft WD-xmlschema11-1-20070830*, 2007.

[42] D. Chamberlin. Xquery: A query language for xml. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 682–682. ACM, 2003.

[43] S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, and J. Richman. {Extensible Stylesheet Language (XSL) Version 1.0}. *W3C Recommendation*, 15, 2001.

[44] J. Clark et al. Xsl transformations (xslt) version 1.0. *W3C recommendation*, 16(11), 1999.

[45] J. Clark, S. DeRose, et al. Xml path language (xpath) version 1.0. *W3C recommendation*, 16:1999, 1999.

[46] A. Eisenberg and J. Melton. An early look at xquery api for java(xqj). *ACM SIGMOD Record*, 33(2):105–111, 2004.

[47] S. Graf, L. Lewandowski, and C. Grün. Distributed systems group database & information systems group jax-rx.

[48] T. Grust. Accelerating xpath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120. ACM, 2002.

[49] Christian Grün, Alexander Holupirek, Marc Kramis, Marc H. Scholl, and Marcel Waldvogel. Pushing XPath accelerator to its limits. In *Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (EXPDB 2006)*. ACM, June 2006.

[50] C. Grün, S. Gath, A. Holupirek, and M. Scholl. Xquery full text implementation in basex. *Database and XML Technologies*, pages 114–128, 2009.

[51] M. Kantrowitz, B. Mohit, and V. Mittal. Stemming and its effects on tfidf ranking (poster session). In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 357–359. ACM, 2000.

[52] OpenEHR. http://www.openehr.org, January 2011.

[53] T. Beale, S. Heard, D. Kalra, and D. Lloyd. The openehr data structures information model, 20.

[54] S. Jones. Toward an acceptable definition of service. *IEEE software*, pages 87–93, 2005.

[55] R.L. Costello. Building web services the rest way. *UR L: http://www. xfront. com/REST-Web-Services. html. Ultima Consulta*, 11:2007, 2007.

[56] S. Graf, L. Lewandowski, and C. Grün. Jax-rx, unified rest access to xml resources. *Technical Report KN-2010-DiSy-01*, 2010.

[57] D. Chamberlin, D. Florescu, J. Robie, et al. Xquery update facility. *W3C working draft*, 8, 2006.

[58] D. Chamberlin, D. Florescu, and J. Robie. Xquery update facility. w3c working draft 11 july 2006, 2007.

[59] B. Bibeault and Y. Katz. jquery in action. *Manning Publications Co. Greenwich, CT, USA*, 2008.

[60] H. Chen and R. Cheng. *ZK: Ajax without JavaScript framework*. Springer, 2007.

[61] E. Burnette. Google web toolkit–taking the pain out of ajax. *USA: The Pragmatic Programmers LLC*, 2006.

[62] F. Daoud. *Stripes:... and Java web development is fun again (Pragmatic Programmers)*. Pragmatic Bookshelf, 2008.

[63] ZK. http://books.zkoss.org, March 2011.